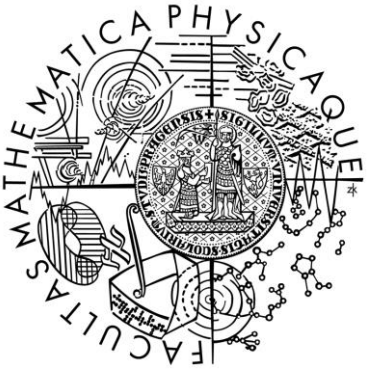
Univerzita Karlova v Praze

Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE



Imrich Kuklis

Vybrané problémy související s vehicle routing

Kabinet software a výuky informatiky

Vedoucí bakalářské práce: RNDr. Pergel Martin, Ph.D.

Studijní program: Informatika

Studijní obor: Obecná informatika

Praha rok 2014

Ďakujem svojmu vedúcemu RNDr. Martinovi Pergelovi Ph.D. za jeho podporu a cenné pripomienky ohľadom textu i obsahu práce.

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona.

V Praze dne 30.07.2014 Imrich Kuklis

Název práce: Vybrané problémy související s vehicle routing

Autor: Imrich Kuklis

Katedra / Ústav: Kabinet software a výuky informatiky

Vedoucí bakalářské práce: RNDr. Martin Pergel, Ph.D., Kabinet software a výuky informatiky

Abstrakt: V této práci prezentujeme a implementujeme několik rozvrhovacích algoritmů. První část pojednává o popisu dopravního problému a její varianty. V druhé části práce popisujeme rozvrhovací algoritmy, které jsme implementovali. V třetí části porovnáme algoritmy podle výsledků různých testů. Další kapitola je věnována dokumentaci. Závěr uvádí možnosti rozšíření bakalářské práce.

Klíčová slova: dopravní problém VRP, časová složitost, algoritmus, celočíselné programování, optimalizace

Title: Particular Problems Related to the Vehicle Routing problem

Author: Imrich Kuklis

Department: Department of Software and Computer Science Education

Supervisor: RNDr. Martin Pergel, Ph.D., Department of Software and Computer Science Education

Abstract: In this thesis we present and implement some scheduling algorithms. In the first part we discuss the vehicle routing problem and its variants. In the second part we describe the algorihms we used in the thesis. In the third part we compare the algorithm by their results on several tests. In the last chapter we describe the documentation of the programs which we implemented and discuss the possible future extension of the thesis.

Keywords: vehicle routing problem (VRP), running time, algorithm, integer programming, optimalization

Názov práce: Vybrané problémy súvisiace s vehicle routing

Autor: Imrich Kuklis

Katedra / Ústav: Kabinet software a výuky informatiky

Vedúci bakalárskej práce: RNDr. Martin Pergel, Ph.D., Kabinet software a výuky informatiky

Abstrakt: V tejto práce prezentujeme a implementujeme niekoľko rozvrhovacích algoritmov. V prvej časti popíšeme dopravný problém a jeho varianty. V druhej časti popíšeme rozvrhovacie algoritmy, ktoré sme implementovali. V tretej časti porovnáme výsledky algoritmov po testovaní. Ďalšia kapitola venuje dokumentácie implementovaných programov. Záver uvádza možnosti rozšírenia práce.

Kľúčové slová: dopravný problém VRP, časová zložitosť, algoritmus, celočíselné programovanie, optimalizácia

Obsah

Úvod

Pre každú spoločnosť, ktorá uplatňuje svoje výrobky na trhu je dôležitý servis spojený transportom výrobkov k odberateľom. Transportné náklady k zákazníkom tvoria značný podiel ceny výrobkov a preto je dôležitá optimalizácia transportu na udržanie konkurencieschopnosti výrobkov.

Ďalším dôležitým faktorom je promptnosť dodávok a dodanie tovaru na dohodnutý termín pri najlepšom možnom využití vlastností kamiónov. Aby poskytnutá služba transportu kamiónmi bola výnosná, firma potrebuje dobrú koordináciu kamiónov. Inými slovami potrebujeme vytvoriť plán pre každý kamión, že v každom momente vieme presne, kde sa nachádza kamión a akú objednávku transportuje do cieľa.

Takou koordináciou sa zaoberá odbor logistika. Pre lepšie pochopenie pojmu logistika zavedieme nasledujúcu definíciu, ktorá je citovaná a prekladaná z anglickej knihy Logistics & Supply Chain Management. „Logistika je proces strategického riadenia verejného obstarávania, pohybu a skladovania materiálov, dielov a hotových zásob (a súvisiace informačné toky) cez organizácie a jeho marketingové kanály takým spôsobom, že súčasná a budúca ziskovosť je maximalizovaná.“[1] V súčasnosti logistika patrí medzi najpopulárnejšie odbory. Má široké uplatnenie v praxi.

V mojej bakalárskej práci sa sústredím na známy problém, s ktorým sa zaoberá veľa vedcov z oblasti logistiky, teoretickej informatiky a z aplikovanej matematiky. Predmetom práce je riešenie problému logistiky ciest čo možno najefektívnejším spôsobom. Zadanie by sa dalo napísať najjednoduchšie takto.

Máme veľkú sieť ciest, ktorá spojí entity(mestá, regióny a štáty). Medzi dvojicami, entitami je obmedzenie, ktoré musíme splniť. Obmedzenie môžeme si predstaviť abstraktne takto, balík obmedzenej veľkosti sa má dostať z mesta A do mesta B. Ako sa dostane balík do cieľového mesta? Pomocou kamiónov, ktoré máme k dispozícií.

Dôležitou otázkou je, či môžeme takýto problém riešiť algoritmicky? Tento problém je známy pod menom *Dopravný problém* (VRP- *Vehicle routing problem*). Cieľom bakalárskej práce je vytvoriť rozvrh pre danú inštanciu problému s rôznymi algoritmami a porovnávať ich výsledky.

V prvej kapitole zadefinujeme základné pojmy informatiky. Medzi základnými pojmami spomenieme *algoritmus, problém, časovú zložitosť, základné triedy časovej zložitosti (P, NP),* ktoré budeme používať v bakalárskej práci. V poslednej podkapitole prvej kapitoly popíšeme najznámejšie problémy triedy NP.

V druhej kapitole sformulujeme znenie problému VRP a popíšeme najznámejšie typy problému VRP. Popíšeme vzťah VRP k lineárnemu programovaniu a k odboru takzvaného *Problém splňovania podmienok* (CSP - constraint satisfaction problem).

V tretej kapitole popíšeme používané algoritmy, ktoré sme implementovali (pseudokód, časová zložitosť, konečnosť, správnosť a stabilitu algoritmu).

V štvrtej kapitole sa sústredíme na testovanie algoritmov s rôznymi inštanciami problému. Skúmame aké výsledky vrátia algoritmy na rôzne testy a špeciálne prípady. Po testovaní popíšeme pozorované výhody a nevýhody algoritmov.

Kapitola 1

Základné pojmy

V tejto kapitole na začiatku zavedieme definíciu *problému.* Pokračujeme potom s definovaním pojmu *algoritmus* a popíšeme najznámejšie triedy *časovej zložitosti P a NP*. Na záver zmienime známe príklady problémov z triedy NP.

1.1 Problém, inštancia problému, algoritmus

Aby sme mohli zaviesť definíciu *algoritmu* a *časovej zložitosti* je nevyhnutné definovať pojem *problému*. Základné pojmy popíšeme podľa knihy Garey & Johnson [2].

1.1.1 Problém

Problém bude všeobecná otázka na ktorú hľadáme odpoveď, obyčajne obsahujúca niekoľko parametrov(voľné premenné), pre ktoré nie sú definované hodnoty dopredu. *Problém* je popísaný pomocou všeobecného popisu všetkých jeho parametrov a prehľad o tom, aké vlastnosti musí splniť odpoveď alebo riešenie. Exituje aj špeciálna kategória problémov takzvané rozhodovacie problémy. Riešenia rozhodovacích problémov majú len dve možnosti **áno** alebo **nie.**

Ďalším dôležitým pojmom je *inštancia problému*, ktorú získame nasledovne.

1.1.2 Inštancia problému

Inštancia problému sa získa pomocou určenia všetkých parametrov *problému* s konkrétnymi hodnotami.

Teraz keď už sme zadefinovali pojem *problém* a *inštancia problému* môžeme zaviesť pojem *algoritmus.*

1.1.3 Algoritmus

Algoritmy všeobecne sú postupy na riešenie problémov krok za krokom. Pre správnosť môžeme ich predstaviť ako počítačové programy napísané v nejakom presnom programovacom jazyku. *Algoritmus* rieši problém Π, pokiaľ rieši ľubovoľnú inštanciu I problému Π a je zabezpečené, že algoritmus vždy spočíta riešenie pre inštanciu I. Algoritmus má ešte dve dôležité vlastnosti, ktoré spomenieme, konečnosť a efektivita algoritmu. Algoritmus je konečný, keď vypočíta inštanciu problému po konečnom počte krokov. Efektivita algoritmu sa delí do dve veľké skupiny, pamäťová a časová zložitosť. U väčšiny algoritmov je dôležitejšia časová zložitosť.

1.2 Časová zložitosť, trieda P a NP

V súčasnosti dôraz u algoritmov je kladený na časovú zložitosť. To neznamená, že je zanedbateľná pamäťová zložitosť. Existuje veľa algoritmov u ktorých ani v dnešnej dobe nemáme dostatočnú pamäťovú kapacitu. Najlepší príklad je na tento algoritmus Minmax pre šachové hry, kde dátová štruktúra algoritmu celej hry sa nezmestí do pamäti žiadneho počítača.

Pre lepšie pochopenie *časovej zložitosti* máme niekoľko základných notácií (**Θ,O,Ω, o, ω**). Vďaka notáciám môžeme popísať asymptotickú časovú zložitosť daného algoritmu ako funkciu s oborom hodnôt definovaných na množine prirodzených čísiel. Pre nás bude najdôležitejšia notácia O. „Pre dané funkcie g(n) označujeme množinu O(g(n)) ako množinu funkcie takto. O(g(n)) = { f(n) : existuje pozitívna konštanta c a n_0, že platí nasledujúci nerovnosť $0 \leq f(n) \leq c * g(n)$ pre každé $n \geq n_0$ } Funkcia f(n) je zhora obmedzená pomocou funkcie g(n).“ [3]. Táto notácia obmedzí časovú zložitosť algoritmu zhora.

Časovú zložitosť môžeme ešte ďalej deliť. V informatike problémy delíme do viacerých základných tried podľa zložitosti. Dve najzákladnejšie a najznámejšie skupiny sú *P* a *NP*. Do množiny P patria také problémy, ktoré môžeme vyriešiť pomocou deterministického Turingového stroja v polynomiálnom čase.

Do triedy P patria také problémy, pre ktoré pre vstup veľkosti n je doba chodu zhora obmedzená s O(n^k), kde k je ľubovoľná konštanta.

Trieda P sa obvykle považuje za takú triedu problémov, ktorú môžeme riešiť efektívne aj pre veľké vstupy dát. Najznámejšie problémy triedy P sú napríklad triedenie čísiel pomocou porovnávania, násobenia matíc, hľadanie najkratších ciest v hranovo ohodnocovanom grafe. V triede P existujú aj také problémy, ktoré síce sú riešiteľné v polynomiálnom čase, ale z praktického hľadiska sú nepoužiteľné, napríklad keď máme časovú zložitosť O(n^{1000}).

Trieda zložitosti NP je taká trieda, ktorá obsahuje také jazyky, ktoré môžeme overiť s algoritmom, ktorý behá v polynomiálnom čase. Presnejšie jazyk L patrí do triedy NP len práve vtedy, keď existuje algoritmus A s dvomi vstupnými parametrami bežiacimi v polynomiálnom čase a existuje taká konštanta c, pre ktorú platí nasledovné: L = {$x \in \{0,1\}^*$: existuje certifikát y taký, že |y| = O((|x|^c) na ktorý platí, že A(x,y) = 1}.

O jazyku L_1 hovoríme, že je *redukovateľný v polynomiálnom čase* na jazyk L_2, označíme $L_1 \leq_P L_2$ existuje funkcia spočítateľná v polynomiálnom čase taká, že $f: \{0,1\}^* \rightarrow \{0,1\}^*$ pre každé $x \epsilon \{0,1\}^*$.

Jazyk $L \subseteq \{0,1\}^*$ je *NP- úplný*, keď platia nasledujúce podmienky:

1. $L \in NP$, a naviac platí,
2. $L' \leq_P L$ pre každé $L' \in NP$

Jazyk $L \subseteq \{0,1\}^*$ je *N -ťažký*, keď platí druhá podmienka z *NP- úplnosti*, ale prvá podmienka nemusí platiť.

1.3 Príklady triedy NP

Do triedy NP patrí niekoľko sto príkladov. My z toho spomenieme niekoľko známych.

1.3.1 Problém obchodných cestujúcich (TSP)

Problém obchodných cestujúcich môžeme popísať nasledovne. Máme úplný graf na n vrcholoch. Každý vrchol je spojený s iným vrcholom práve raz, slučky nie sú povolené a každá hrana má nezápornú váhu.

V grafe je prítomný obchodný cestujúci, ktorý chce navštíviť všetky vrcholy v grafe za podmienky:

1. Každý vrchol môžeme navštíviť iba raz.
2. Musíme skončiť v tom istom mieste, kde sme začínal svoju cestu.

Exituje aj rozhodovacia verzia problému TSP. Formálny popis rozhodovacieho problému TSP:

TSP={$\langle G, c, k \rangle$, kde G(V,E) je úplný graf na n vrcholoch, c je nasledovná funkcia $c: Vx\ V \rightarrow Z$, $k \in Z$ a graf G má kružnicu veľikosti obmedzená zhora s k}.

1.3.2 Kľučka

Nech máme neorientovaný graf G (V, E), kde V je množina vrcholov a E je množina hrán a nech máme $V' \subseteq V$ podmnožinu vrcholov, kde každá dvojica vrcholov ja spojená s hranou z množiny E sa nazýva *kľučka* (CLIQUE). Inými slovami kľučka je úplný podgraf grafu G. Veľkosť kľučky je počet vrcholov ktoré obsahuje. Problém kľučky je problém optimalizácie, ktorá hľadá maximálnu kľučku v danom grafe. Rozhodovací problém CLIQUE = {$\langle G, k \rangle$, kde G je graf s kľučkou veľkosti k}.

1.3.3 Vrcholové pokrytie (VERTEX-COVER)

Nech máme graf G(V,E) a nech máme $V' \subseteq V$ podmnožinu vrcholov, takú na ktorú platí keď $(u, v) \in E$ potom buď $u \in V'$ alebo $v \in V'$ alebo nastanú obidva prípady. Každé vrcholové pokrytie pokrýva susediace hrany daného vrcholu. Vrcholové pokrytie grafu G je množina takých vrcholov, ktoré pokrývajú všetky hrany grafu G. Veľkosť vrcholového pokrytia sa rovná k počtu uzlov v ňom. Úlohou problému vrcholového pokrytia je nájsť najmenšie vrcholové pokrytie grafu G.
Tento optimalizačný problém môžeme preformulovať na rozhodovací problém takto: Hľadáme vrcholové pokrytie grafu G veľkosti k. VERTEX-COVER = $\{\langle G, k\rangle$, kde G je graf s vrcholovom pokrytím veľkosti k$\}$. Vrcholové pokrytie je NP - úplný problém.

1.3.4 Súčtová podmnožina

Problém súčtovej podmnožiny je aritmetický problém. V probléme súčtovej podmnožiny máme danú konečnú podmnožinu $S \subset N$ a cieľovú hodnotu $t \in N$. Otázkou problému, či exituje taká podmnožina čísiel $S' \subseteq S$, že ich súčet sa rovná hodnote t. Súčtová podmnožina je NP - úplný problém.
Jeden z najznámejších problémov informatiky je problém, či sa množina P rovná množine NP. Zatiaľ vieme len, že platí $P \subseteq NP$. Tento problém patrí do kategórie tisícročných problémov.

Kapitola 2

VRP

Na začiatku tejto kapitoly popíšeme dopravný problém VRP. Po popise problému VRP nasleduje vymenovanie niekoľko typov problémov VRP. Ďalej popíšeme vzťah VRP k odboru CSP a lineárneho programovania. V rámci lineárneho programovania sústredíme sa na jeho špeciálny odbor zmiešaného celočíselného programovania.

2.1 Popis problému VRP

VRP sa zaoberá spravovaním vyzdvihnutia tovaru a dodávaním vyzdvihnutého tovaru k cieľovej osobe alebo firme. Hlavnou otázkou VRP je ako využívať dostupnú skupinu kamiónov, aby sme mohli čo najúčinnejším spôsobom vyhovieť požiadavkám dopytovaných služieb.

Problém VRP môžeme formulovať nasledovne. Máme obecný graf, ktorý môže obsahovať orientované a neorientované hrany. Úlohou VRP je nájsť takú množinu ciest, ktorá obslúži objednávky všetkých zákazníkov a ich nákladová funkcia množiny ciest je minimálna. Nákladová funkcia z anglického slova cost je špeciálna funkcia, ktorá priradí k ceste kamiónu reálne číslo.

VRP sa delí na dve veľké skupiny problémov statické a dynamické problémy.

U statických problémov požiadavky na služby sú dopredu známe a nemienia sa behom vykonania daných služieb. Ďalším dôležitým predpokladom je, že kamióny zabezpečujú rovnaký druh služby. U dynamických služieb môžeme dostať požiadavky aj behom vykonania služieb. Pre nás budú dôležité statické problémy.

VRP sa skladá z nasledujúcej pätice, z mapy ciest a miest, z objednávok, z kamiónov, lokálnych a globálnych podmienok .

2.2 Modelovanie VRP, globálne a lokálne podmienky

Mapu VRP modelujeme pomocou grafov. Graf reprezentuje sieť ciest a miest. Hrany grafu sú cesty, ktoré majú dve vlastnosti, ktoré sú dôležité z pohľadu modelovania.

Prvá vlastnosť je smer cesty. Cesty môžu byť jednosmerné alebo obojsmerné.

Druhá vlastnosť je dĺžka cesty, ktorú môžeme reprezentovať pomocou reálnych čísiel , ktoré sú väčšie alebo rovné nule.

U obojsmerných ciest dĺžka cesty v oboch smeroch je rovnaká. Mestá sú križovatky medzi cestami, ktoré môžu byť garážou kamiónu, miestom objednávky alebo miestom zákazníka.
Kamióny transportujú objednávky k zákazníkom. Pri modelovaní môžeme ich popísať mnohými rôznymi vlastnosťami. Medzi najdôležitejšie vlastnosti patrí rýchlosť, kapacita, rozmery skladovacieho priestoru, cena za jednotku vzdialenosti, spoľahlivosť.
Kamión vykonáva len dve úlohy, vyzdvihne tovar a transportuje k zákazníkovi.
U objednávkach medzi najdôležitejšie informácie patria, mesto odkiaľ musíme transportovať, mesto kam musíme transportovať, množstvo a rozmery tovaru, priorita a čas dodávky.
Pri modelovaní problému VRP používame ešte lokálne a globálne podmienky.
Lokálne podmienky sú také podmienky, ktoré sú platné len na nejakú časť modelu, ale nie na celý model. Príkladom lokálnej podmienky môže byť napríklad podmienka, že kamión s identifikačným číslom 111 pri transporte musí mať vždy aspoň polovicu nakladacieho priestoru naplnené.
Globálne podmienky sú také podmienky , ktoré sú platné na celý model. Príkladom môže byť minimalizácia súčtu výdavkov na transportovanie. Vo väčšine prípadov lokálne a globálne podmienky sú nezlučiteľné. Môže nastať, ak optimalizujeme tak, aby sme vyhoveli čo najlepšie pre danú lokálnu podmienku porušíme s tým nejakú globálnu podmienku. Preto pre podmienky môžeme vytvoriť priority, takzvaný preferovací zoznam. Musíme sa rozhodnúť, že ktoré podmienky budeme preferovať. Pri preferovaní by sme mali vybrať také podmienky, ktoré nesmú byť protichodné . Keď nevytvoríme preferovací zoznam tak globálne podmienky automaticky majú väčšiu prioritu. Cieľom modelovania je lepšie pochopenie problému a lepšie predstavenie korelácií.

2.3 Základné varianty VRP

Problém VRP delíme do dvoch hlavných skupín podľa potrieb zákazníka *uzlovo dopravný problém (NRP - node routing problem)* a *hranovo dopravný problém (ARP- arc routing problem).* U ARP zákazníci požadujú rovnakú distribúciu (tovaru, objednávok) medzi hranami grafu, vrcholy sú nepodstatné pre ARP. U NRP zákazníci sa nachádzajú v mestách. My sa budeme zaoberať problémom NRP. NRP

sa často označuje aj ako vehicle scheduling problem. NRP patrí do skupiny NP - ťažké problémy. Bez globálnych a lokálnych podmienok sa dá redukovať na problém TSP, ktoré sme zmienili na konci prvej kapitoly.

Pri redukovaní VRP na TSP rozšírime graf G problému VRP na úplný graf G'.

Úplný graf ja taký graf, kde počet hrán je $\binom{n}{2}$, kde n je počet vrcholov v grafe.

V úplnom grafe každá rôzna dvojica vrcholov je spojená hranou . Z množiny vrcholov vyberieme jeden vrchol, ktorý budeme označovať ako depo kamiónu , ostatné vrcholy sú zákazníci. Objednávky doručujeme len jediným kamiónom. Kamiónu dovolíme nekonečnú kapacitu, aby mohol transportovať všetky objednávky súčasne. Cieľom je navštíviť všetky vrcholy grafu a doručovať všetky objednávky tak, že kamión začne svoju cestu z depa, každého zákazníka navštívi len raz, končí svoju cestu v depe a absolvovaná cesta má minimálne náklady. Táto cesta s minimálnym nákladom sa menuje minimálna Hamiltonovská kružnica.

Hamiltonovská kružnica v grafe G je usporiadanie V(G) = $\{v_1, v_2, \dots, v_n\}$ také, že pre $\forall\, i:\ v_i v_{i+1} \in E(G)$ a na viac platí, že $v_1 = v_n$.

Problém TSP rozšírime na problém multi-TSP (MTSP). MTSP sa líši od TSP s tým, že máme k dispozícií n kamiónov a každý kamión musí obsluhovať aspoň jedného zákazníka. Riešením MTSP sú n Hamiltonovské kružnice s minimálnym nákladom, také že každý zákazník je obslúžený len jedným kamiónom.

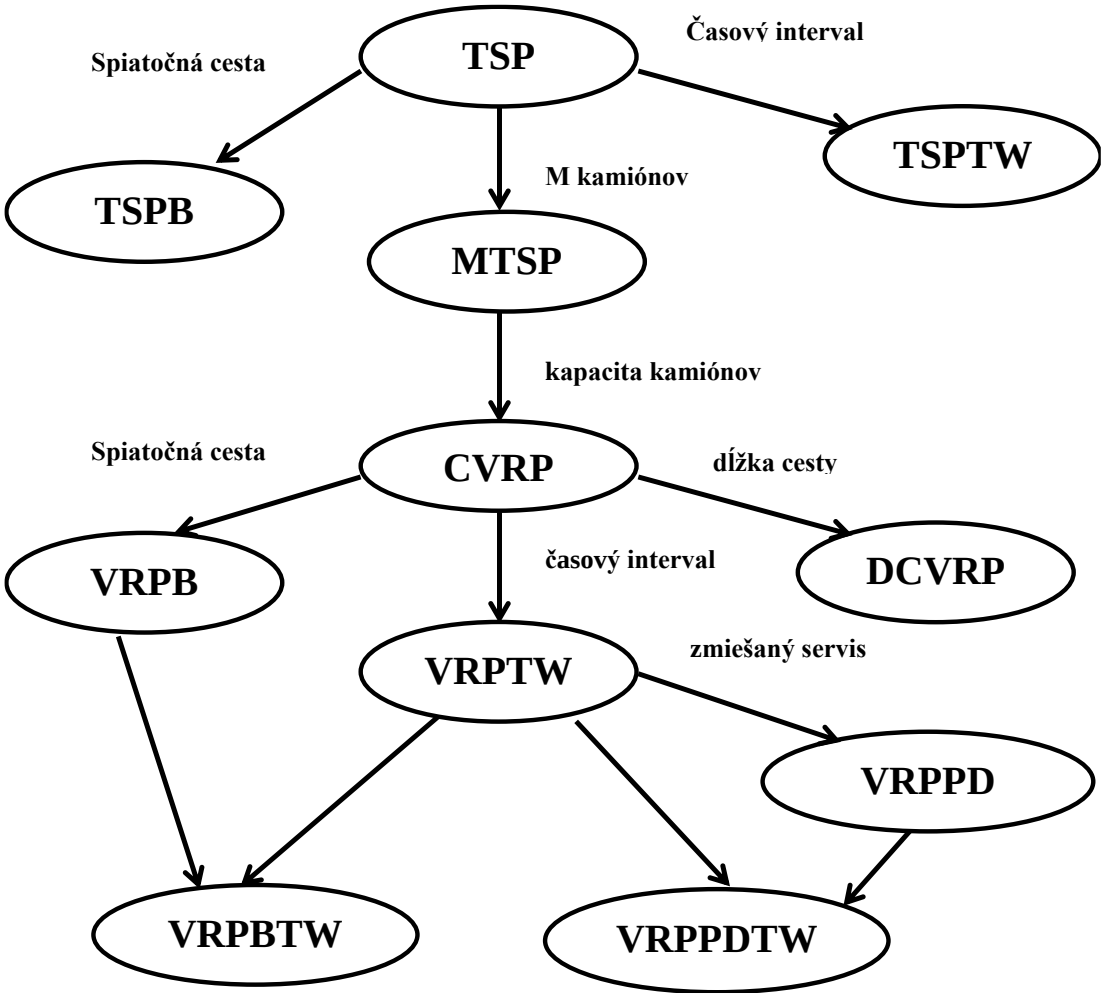
Problém MTSP môžeme rozšíriť na kapacitné VRP (CVRP). Rozdielom medzi MTSP a CVRP je, že už nemáme dovolenú nekonečnú kapacitu. CVRP obsahuje *m* identických kamiónov s kapacitou veľkosti *c*. Ďalšie obmedzenie je, že veľkosť objednávok zákazníkov nepresahuje kapacitu kamiónov. Známa modifikácia CVRP je VRP s podmienkami na vzdialenosť (DCVRP), kde riešenie problému je Hamiltonovská kružnica s minimálnou vzdialenosťou. Existuje aj špeciálny prípad CVRP takzvané VRP with backhaul (VRPB). Tento problém sa líši s tým od ostatných VRP problémov, že v tomto prípade zákazníkov rozdelíme na dve veľké skupiny. Prvá skupina zákazníkov Linehaul Customers (LC) potrebuje požadované množstvo tovaru dodané. Druhá skupina zákazníkov Backhaul Customers (BC) potrebuje požadované množstvo tovaru naložiť. Podmienka na ceste je, že každý LC má prednosť pred BC.

VRP s časovým okienkom (VRPTW) zovšeobecňuje CVRP. V tomto prípade každý zákazník má svoj časový interval. Pre službu presne máme určené, že kedy môžeme

začať vykonať službu a dokedy musíme dokončiť. Keď kamión dorazí pred daným časovým intervalom musí na danom mieste čakať dovtedy, kým môže začať službu. Objednávku môžeme doručiť len behom časového intervalu. Cesty sa štartujú v čase 0. Riešením problému VRPTW je množina *k* ciest s minimálnym nákladom. Na cesty platia nasledujúce podmienky: Každá cesta musí navštíviť depo. Každý zákazník je navštívený len s jednou cestou. Súčet množstva objednávok na ceste nemôže presiahnuť kapacitu kamiónu. Pre každého zákazníka vykonáme servis len v časovom intervale, ktorý zadefinoval zákazník.

Posledný typ VRP, ktorý popíšeme je VRP s dodaním a naložením (VRPPD). V tomto prípade zákazník má objednávky z obidvoch skupín LC a BC.

Vzťah medzi základnými VRP problémami je na obr.1.



Obrázok č.1. : Vzťah medzi základnými VRP problémami

Doteraz sme modelovali pomocou grafov. V zostávajúcej časti druhej kapitoly ukážeme aj iné modelovacie techniky, aby sme mali lepšiu predstavu o probléme z rôznych hľadísk.

2.4 Modelovanie pomocou CSP

V podkapitole 2.2 sme zaviedli pojem globálnych a lokálnych podmienok.
V teoretickej informatike existuje problém, ktorý sa špecializuje na modelovanie problémov s podmienkami. Tento problém sa menuje CSP.[5]
Hlavnou úlohou CSP je sústrediť sa čo najpresnejšie na modelovanie problému, viac ako na spôsob riešenia problému a reprezentovať korelácie problému, *CSP* sa skladá z konečných množín *premenných*, *domény premenných* a konečných množín *podmienok*.
Premenné popisujú charakteristiku riešenia o ktorom je potrebné sa rozhodnúť. *Doménu* premennej môžeme predstaviť ako konečnú množinu hodnôt. V niektorých prípadoch sú dovolené aj také premenné, ktoré majú množinu nekonečných hodnôt. Pre nás budú dôležité premenné s konečnými množinami hodnôt. U niektorých problémov existuje takzvaná superdoména, ktorá je spoločná pre všetky premenné a jednotlivé domény sa vytýči pomocou unárnych podmienok.
Unárna podmienka je podmienka s jedným parametrom. *Podmienka* je ľubovoľná relácia nad množinou premenných. Podmienku môžeme definovať ako množinu kompatibilných n-tíc alebo ako formulu.
Problém NRP sa dá modelovať pomocou CSP. U NRP globálnou podmienkou bude minimalizácia nákladovej funkcie NRP. Premenné problému sú kamióny. Doména kamiónu je postupnosť *k*-tíc, kde každá *k*-tica reprezentuje postupnosť *k* objednávok obslúžené behom cesty kamiónu. Cieľom je nájsť hodnotu pre všetky premenné tak, aby nákladová funkcia bola minimálna a aby sme vyhoveli čo najlepšie pre ostatné podmienky z množiny podmienok. Pomocou podmienok z množiny podmienok môžeme vyfiltrovať nedovolené hodnoty z domény premenných. Výhoda filtrovania domén je, že keď dostaneme premenné s prázdnou doménou potom inštancia problému VRP nemá riešenie. V ostatných prípadoch musíme vyskúšať všetky možné n-tice k-tíc aby sme našli najlepšie možné riešenie. Obecne množina n-tíc má exponenciálnu kardinalitu. *Kardinalita* je číslo reprezentujúce počet prvkov v množine.

2.5 Modelovanie pomocou lineárneho programovania (LP)

Ďalší odbor, ktorý sa zaoberá modelovaním a riešením NP- ťažkých a NP- úplných problémov je odbor LP, ktorý je špeciálnym odborom optimalizácie.

Cieľ LP by sme mohli vysvetliť nasledujúcim spôsobom. „V obecnej úlohe lineárneho programovania hľadáme vektor $x^* \in R^n$ maximalizujúci alebo minimalizujúci hodnotu danej lineárnej funkcie medzi všetkými vektormi $x \in R^n$ spľnujúci sústavu lineárnych rovníc a nerovníc. Lineárna funkcia, ktorá sa maximalizuje (minimalizuje) sa nazýva *účelová funkcia* a má tvar $c^T x = \sum_{i=1}^{n} c_i * x_i$, kde $c \in R^n$ je daný vektor.“[4] Vektor $x \in R^n$, ktorý splňuje všetky obmedzenia danej úlohy sa nazýva *prípustné riešenie*. Každému $x^* \in R^n$, ktorý dáva najväčšiu možnú hodnotu účelovej funkcie sa nazýva *optimálne riešenie*.

Keď sme už zaviedli základné pojmy LP môžeme formulovať dopravný problém VRP y hľadiska LP.

Dopravný problém sa skladá z výrobcov a spotrebiteľov. Výrobcov označíme takto $V_1, \dots, V_m$, vyrábajú výrobok *v* v množstvách $a_1, \dots, a_m > 0$. Spotrebiteľov označíme $S_1, \dots, S_n$, konzumujúci výrobok *v* v množstvách $b_1, \dots, b_n > 0$. Podmienkou ekonomickej rovnováhy je $\sum a_i = \sum b_j$. Táto rovnosť nám dovoľuje pridať fiktívny S alebo V.

Premenné $x_{ij} \geq 0$ reprezentujú množstvo výrobkov *v* vyrábaných i-tým výrobcom a transportované k j-tému spotrebiteľovi. Premenné $c_{ij} \geq 0$ reprezentujú náklad na dopravu jedného výrobku transportovaného i-tým výrobcom k j-tému spotrebiteľovi.

Formulácia klasického dopravného problému: $\min \sum_i \sum_j c_{ij} * x_{ij}$ na množine *M* za podmienky $\sum_{i=1}^{m} x_{ij} = b_j \; pre \; \forall j = 1, \dots, m$, $\sum_{j=1}^{n} x_{ij} = a_i \; pre \; \forall i = 1, \dots, n$ a $x_{ij} \geq 0 \; pre \; \forall i, j.$

Celočíselné programovanie od lineárneho programovania je odlišný tým, že pre premenné x_{ij} dovoluje len celočíselné hodnoty. Podľa literatúry : „Je obecne známa, že obecná úloha celočíselného programovania je algoritmicky ťažká (presne povedané NP- úplná), na rozdiel od úlohy lineárneho programovania. Pridaním podmienky celočíselnosti sa drasticky zmení obtiažnosť úlohy.“ [6]

Medzi najznámejšie príklady celočíselného programovania patrí TSP, klasický dopravný problém, párovanie maximálnej váhy.

V praxi sa používa LP relaxácia so slabšími požiadavkami, ktorá dovolí neceločíselné hodnoty .

Výhodou LP relaxácie je ,že dá horný odhad na najlepšie možné riešenie pôvodnej úlohy.

Kapitola 3

Implementácia algoritmov

Na začiatku tejto kapitoly zadefinujeme presný model nášho problému. Ďalej popíšeme algoritmy, ktoré sme implementovali. U každého algoritmu popíšeme stručný pseudokód algoritmu. Každý dôležitejší príkaz pseudokódu má vlastný riadok. Rozoberieme príkazy pseudokódu a vysvetlíme ich význam. Popíšeme charakteristiky algoritmu. Medzi charakteristikami sa sústredíme na konečnosť, časovú zložitosť a determinizmus algoritmu.

3.1 Model problému

V predchádzajúcej kapitole sme popísali rôzne typy problému VRP. U každého typu problému sme sa sústredili na jeho špeciálne vlastnosti. Spoločnou vlastnosťou každého typu VRP je minimalizovať nákladovú funkciu.
Náš model sa skladá z miest, ciest, kamiónov a objednávok.
Mestá sú vrcholmi grafu. Mesto môžeme reprezentovať v modeli pomocou prirodzených čísiel. Každé mesto má svoje vlastné unikátne prirodzené číslo, ktoré je identifikačným číslom mesta.
Cesty reprezentujú hrany grafu. Náš graf obsahuje len neorientované hrany.
V skutočnosti každú neorientovanú hranu reprezentujeme s dvomi orientovanými hranami. Každá orientovaná hrana má svoje vlastné identifikačné číslo. Jediná vlastnosť cesty, ktorá je pre nás dôležitá je dĺžka cesty. Dĺžku cesty reprezentujeme pomocou kladného reálneho čísla. Orientované hrany , ktoré reprezentujú neorientovanú cestu majú rovnakú dĺžku. V modeli nie je dovolená taká hrana, ktorá sa začína a končí v tom istom meste.
Kamióny reprezentujeme pomocou pätice. Do pätice patria nasledujúce vlastnosti, identifikačné číslo kamiónu, identifikačné číslo mesta odkiaľ začína vykonať svoju prácu kamión, kapacita, rýchlosť a cena za jednotku vzdialenosti. V našom prípade kamióny nemusia dodržať takú silnú podmienku aká je popísaná v problému CVRP. Kamióny môžu mať rôznu kapacitu, rýchlosť a cenu za jednotku vzdialenosti. Kapacita, rýchlosť a cena je reprezentovaná pomocou kladného reálneho čísla.
Objednávka v našom modeli reprezentuje transportnú službu tovaru z mesta skladu tovaru do mesta zákazníka. V modeli dovoľujeme rozobrať tovar na menšie množstvá. Objednávky reprezentujeme pomocou pätice. Päticu tvorí identifikačné

číslo objednávky, identifikačné číslo mesta skladu tovaru, mesto zákazníka, množstvo tovaru a spracované množstvo tovaru. Spracované množstvo tovaru je tá časť tovaru, ktorá je na ceste k zákazníkovi alebo dodané zákazníkovi.
Množstvo tovaru a spracované množstvo tovaru sú reprezentované kladnými reálnymi číslami.
Informácie v modeli môžeme rozdeliť na dve veľké skupiny, statické a dynamické informácie. Do skupiny statických informácií patria také informácie, ktoré sú konštantné v celom modeli, informácie ktoré sa nikdy nezmenia v inštancií problému. Dynamické informácie sú také informácie, ktoré s časom sa menia.
Naším cieľom je vytvoriť taký rozvrh pre kamióny, aby všetky objednávky boli vybavené čo možno v najkratšom čase.
Všetky naše algoritmy pri vytvorení rozvrhu pre kamióny používajú techniku dynamického rozvrhovania. V našej práci dynamické rozvrhovanie znamená ,že vytvoríme rozvrh len pre tie kamióny, ktoré sú v danom momente sú voľné.

3.2 Náhodný algoritmus

Prvý algoritmus, ktorý popíšeme je nedeterministický algoritmus. „*Algoritmus* je *nedeterministický*, ak jeho správanie sa môže líšiť medzi spusteniami.“[2]. Pokiaľ algoritmus je nedeterministický potom existuje nejaký stav, ktorého nasleduje niekoľko rôznych stavov z ktorých algoritmus vyberie náhodne.
Idea prvého algoritmu je celkom jednoduchá. Vytvoríme poradie objednávok podľa ich identifikačných čísiel. Pre každý voľný kamión, ktorý máme k dispozícií vygenerujeme náhodné číslo z intervalu [0, počet objednávok - 1]. Čísla z intervalu reprezentujú poradové čísla objednávok. Toto číslo nám špecifikuje, že ktorú objednávku obslúži voľný kamión.

3.2.1 Pseudokód algoritmu

1. **While** existuje objednávka, ktorá nie je spracovaná **do**
2. **begin**
3. **If** existuje voľný kamión **then** vyberieme prvý voľný
4. **else** čakáme dovtedy, kým sa stane niektorý kamión voľným,
5. vygenerujeme náhodnú objednávku pre vybraný kamión
6. kamión transportuje objednávku
7. kamión, ktorý sa dostal do cieľa s objednávkou, sa stane voľným,

8. opakujeme krok 3.-7.
9. **end**

3.2.2 Popis príkazov pseudokódu

Pred spustením algoritmu musíme vyfiltrovať nevhodné hodnoty z parametrov algoritmu. Ktoré hodnoty parametrov nie sú vhodné? Nevhodné hodnoty parametrov môžeme rozdeliť do dvoch veľkých skupín, nevhodné transportné objednávky a nevhodné kamióny. Nevhodná transportná objednávka je taká objednávka, ktorú nemôžeme transportovať zo skladu k zákazníkovi. Medzi hlavnými dôvodmi netransportovateľnosti objednávky patrí neexistencia cesty medzi skladom a zákazníkom v grafe a neexistencia kamiónu, ktorý môže obslúžiť objednávku.
V prípade druhej možnosti graf problému je nesúvislý. Kamión je nevhodný pokiaľ nemôže obslúžiť žiadnu objednávku.
Algoritmus musí bežať dovtedy ,kým existuje objednávka, ktorá nie je spracovaná.
V prvom riadku otestujeme, či existuje ešte taká objednávka, ktorá nie ja spracovaná.
V treťom a štvrtom riadku čaká algoritmus dovtedy ,kým nemá voľný kamión. Kamión sa stane voľným keď obslúžil objednávku alebo časť objednávky.
V piatom riadku algoritmus generuje náhodné poradové číslo pre voľný kamión. Toto číslo špecifikuje, ktorú objednávku obslúži voľný kamión.
V šiestom riadku kamión transportuje objednávku k zákazníkovi. Aby sme mohli transportovať musíme hľadať cestu medzi kamiónom a skladom objednávky a cestu medzi skladom objednávky a mestom zákazníka. V tejto úlohe nám pomôže algoritmus Floyd-Warshall. Tento algoritmus hľadá najkratšie cesty medzi všetkými dvojicami miest v grafe. Časová zložitosť tohto algoritmu je $O(n^3)$, kde n je počet vrcholov v grafe. Algoritmus opakuje kroky 3-7, kým nie sú všetky objednávky spracované.

3.2.3 Konečnosť algoritmu

V každom cyklu algoritmu množstvo niektorej objednávky sa zmenší. Súčet množstiev objednávok je konečný. Tieto pozorovania implikujú, že po konečnom počte krokov sa súčet zmenší na nulu z čoho vyplýva, že algoritmus je konečný.

3.2.4 Časová zložitosť algoritmu

Aby sme mohli určiť časovú zložitosť celého algoritmu , musíme určiť časovú zložitosť každého jednotlivého hlavného podkroku. Na začiatku algoritmu musíme hľadať všetky najkratšie cesty v grafe. Táto operácia trvá $O(n^3)$, kde n je počet miest

v grafe. Ďalšou inicializačnou operáciou je filtrovanie nevhodných vstupných parametrov. Filtrovanie objednávok a kamiónov trvá $O(k * m)$, kde k je počet kamiónov a m je počet objednávok.
Kroky algoritmu v treťom až v piatom riadku trvajú $O(1)$ čas, takzvaný konštatný čas. Operácia v šiestom riadku môže trvať až l krokov, kde l reprezentuje počet hrán v grafe, pretože dokážeme vytvoriť graf , v ktorom všetky hrany tvoria najkratšie cesty medzi dvoma vrcholmi grafu. Operácia v siedmom riadku tiež trvá konštantný čas. Posledná otázka, ktorá nám zostala je, že koľkokrát sa opakuje cyklus algoritmu.
Teoreticky najhorší prípad, ktorý môže nastať je ten ,keď každú objednávku obslúži kamión s minimálnou kapacitou. Z tohto pozorovanie dostaneme, že horný odhad na počet cyklov je $p = \sum_{i=1}^{n} \left\lceil \frac{m_i}{c} \right\rceil$, kde p je počet cyklov, ktorý algoritmus vykoná, m_i je množstvo tovaru v i-tej objednávke, c je kapacita kamiónu, ktorý má minimálnu kapacitu a n je počet objednávok. Celková časová zložitosť, ktorú dostaneme je $O(n^3 + k * m + p * l)$.

3.3 Modifikovaný algoritmus dámska a pánska volenka

Problém VRP môžeme si predstaviť ako rozvrhovací problém. Pre každý kamión hľadáme rozvrh tak, aby rozvrh čo najlepšie vyhovel všetkým podmienkam inštancie problému. Presnejšie hľadáme postupnosť rozvrhovacích párov.
Druhý algoritmus, ktorý som použil v bakalárskej práci sa zaoberá problémom stabilného párovania. Na tento problém existuje algoritmus bežiaci v polynomiálnom čase, vymysleli ho pán Galey a Shape [7]. Algoritmus v slovenskej literatúre je známy pod menom pánska volenka. Vstupom algoritmu sú dve množiny, ktoré majú rovnakú veľkosť. Prvá množina obsahuje mužov a druhá ženy. Každý muž má svoj preferovací zoznam žien a tiež každá žena má svoj preferovací zoznam mužov. Cieľom algoritmu je nájsť stabilné párovanie. Je dokázané, že algoritmus vždy nájde stabilné párovanie. Nevýhoda algoritmu je, že nájde párovanie len rovnako veľkých množín. Existuje algoritmus aj na obecnejšiu verziu problému, keď množiny majú rôzne veľkosti, ale je dokázané, že neexistuje stabilné párovanie pre všetky inštancie problému.
U problému VRP tiež môžeme hľadať stabilné párovanie. U toho problému prvá množina reprezentuje voľné kamióny a druhá množina reprezentuje objednávky,

ktoré čakajú na obsluhu. Problémom je, že u väčšiny prípadov veľkosť týchto dvoch množín je rôzna. Potrebujeme modifikovať algoritmus tak, aby vždy pracoval len s rovnako veľkými množinami. Tento cieľ môžeme dosiahnuť pomocou filtrovania množín.

Pri modifikácií algoritmu som rozdelil vstup na päť rôznych možností.

Prvá možnosť je, keď máme len jeden voľný kamión. V tomto prípade vyberieme prvú objednávku z preferovacieho zoznamu kamiónu a máme hotové párovanie.

Druhá možnosť je, keď máme jednu objednávku a *n* kamiónov. Vytvoríme páry podľa preferovacieho zoznamu objednávok. Tretia možnosť je, keď máme viac kamiónov ako objednávok. V tomto prípade musíme filtrovať množinu kamiónov.

Štvrtý prípad je opačným prípadom tretieho a musíme filtrovať množinu objednávok.

Posledná možnosť je ideálny stav v ktorom môžeme používať algoritmus pánska volenka na vytvorenie stabilného párovania.

Algoritmus nechá možnosť voľby pre programátora ako vytvoriť preferovacie zoznamy kamiónov a preferovacie zoznamy objednávok. Ďalšou voľnou možnosťou algoritmu sú filtrovacie algoritmy množín.

3.3.1 Pseudokód algoritmu

1. **While** existuje objednávka, ktorá nie je spracovaná **do**
2. **begin**
3. vytvor preferovacie zoznamy voľných kamiónov
4. vytvor preferovacie zoznamy objednávok čakajúcich na spracovanie
5. **if** počet kamiónov = 1 **then** zavolaj funkciu special truck a skoč na krok 10
6. **if** počet objednávok = 1 **then** zavolaj funkciu special package a skoč na krok 10
7. **if** počet objednávok > počet voľných kamiónov **then** vyfiltruj objednávky.
8. **if** počet voľných kamiónov > počet objednávok **then** vyfiltruj kamióny.
9. vykonaj algoritmus stabilného párovania.
10. kamióny transportujú objednávky k zákazníkovi
11. kamión, ktorý sa dostal do cieľa s objednávkou, sa stane voľným,
12. Opakujeme krok 3-11
13. **end.**

3.3.2 Popis príkazov pseudokódu

Algoritmus beží dovtedy, kým existuje taká objednávka, ktorá nie je spracovaná. Túto podmienku reprezentuje prvý riadok pseudokódu.

V treťom riadku vytvoríme preferovacie zoznamy jednotlivých kamiónov. Preferovacie zoznamy vytvoríme podľa vzorca $h = \frac{t}{r*c}$, kde $t = t_1 + t_2$, kde t_1 reprezentuje množstvo tovaru transportované u prvej objednávky a t_2 transportované množstvo tovaru transportované u druhej objednávky a tieto dve objednávky sú vykonané po sebe. Parameter $r = r_1 + r_2 + r_3 + r_4$, kde r_1 reprezentuje dĺžku cesty od lokality daného kamiónu ku skladu prvej objednávky, r_2 je dĺžka cesty od skladu prvej objednávky k prvému zákazníkovi, r_3 reprezentuje dĺžku cesty medzi zákazníkom prvej objednávky a skladom tovaru druhej objednávky a r_4 reprezentuje dĺžku cesty medzi skladom druhej objednávky a zákazníkom druhej objednávky. Parameter c reprezentuje cenu za jednotku vzdialenosti daného kamiónu. Tento vzorec vypočítame pre všetky možné dvojice objednávok a uložíme do matice typu $n * n$, kde n je počet objednávok. Z každého riadku matice vyberieme najlepšiu hodnotu a uložíme do poľa. Triedime hodnoty poľa vzostupne a vložíme objednávky do preferovacieho zoznamu kamióna s klesajúcim poradím.

V štvrtom riadku vytvoríme preferovací zoznam objednávok. Preferovací zoznam vytvoríme podľa vzorca $h = \frac{d*c}{k}$, kde $d = d_1 + d_2$, kde d_1, je vzdialenosť kamiónu od skladu objednávky a d_2 je vzdialenosť skladu objednávky od zákazníka, c je cena za jednotku vzdialenosti u daného kamiónu a k je kapacita daného kamiónu. Tento vzorec aplikujeme na každý kamión. Zoznam triedime vzostupne.

V piatom riadku vytvoríme rozvrh pre jediný kamión. Vyberieme pre kamión prvú objednávku z preferovacieho zoznamu. Táto objednávka je najpreferovanejšia objednávka pre daný kamión. Po dokončení skočíme na príkaz v desiatom riadku.

V šiestom riadku vytvoríme rozvrh kamiónov podľa preferovacieho zoznamu poslednej nespracovanej objednávky.

V siedmom riadku vyfiltrujeme objednávky, ktoré nemajú kamión, ktorý ich obslúžil. Pre každú objednávku vypočítame súčet ich poradia. Triedime objednávky podľa súčtu a vyberieme prvých *k* najmenších, kde *k* reprezentuje počet kamiónov, ktoré môžu obslúžiť objednávky.

V ôsmom riadku vyfiltrujeme kamióny, ktoré nemajú objednávku, ktorú by mohli transportovať. Pre každý kamión vypočítame súčet ich poradia. Triedime kamióny podľa súčtu a vyberieme prvých *k* najmenších, kde *k* reprezentuje počet objednávok , ktoré ešte nie sú spracované.

V deviatom riadku spustíme algoritmus pánska volenka na kamióny a objednávky, ktoré zostali po filtrovaní. Desiaty a jedenásty riadok vykonáva rovnaké operácie ako riadok šesť a sedem v náhodnom algoritme.

3.3.3 Konečnosť algoritmu modifikovanej pánskej a dámskej volenky

Rovnakým argumentom ako u náhodného algoritmu sa dá dokázať, že algoritmus je konečný. V každom cykle algoritmu množstvo tovaru niektorej objednávky alebo niektorých objednávok sa zmenší. Nakoľko súčet množstiev tovaru objednávok je konečný aj počet vykonaných cyklov je konečný. Rozdiel medzi náhodným algoritmom a týmto algoritmom je, že tento algoritmus počíta deterministicky.

3.3.4 Časová zložitosť algoritmu modifikovanej pánskej a dámskej volenky

Keď chceme určiť časovú zložitosť algoritmu musíme určiť časovú zložitosť jednotlivých krokov.

Príkaz tretieho riadku má časovú zložitosť $O(k * n^2)$, kde k je počet voľných kamiónov a n je počet objednávok čakajúcich na spracovanie. Príkaz štvrtého riadku má časovú zložitosť $O(n * k * \log k)$. V piatom riadku príkaz má časovú zložitosť v najhoršom prípade $O(n)$ pretože, keď sme už transportovali poslednú časť objednávky potom objednávka sa stane spracovanou a musíme zmazať z poľa objednávok. Príkaz v šiestom riadku má časovú zložitosť $O(k)$. To nastane práve vtedy, keď všetky voľné kamióny môžu transportovať časť poslednej objednávky. V šiestom riadku príkaz má časovú zložitosť $O(n * k + n \log n + (n - k) * n)$. V siedmom riadku časová zložitosť je podobná ako v šiestom riadku $O(n * k + k \log k + (k - n) * k)$. Časová zložitosť v ôsmom riadku je $O(p^2)$, kde $p = \min(n, k)$. Časová zložitosť deviateho a desiateho riadku je rovnaká ako u náhodného algoritmu, príkaz na riadku šesť a sedem. Počet cyklov, ktoré vykonáva algoritmus je rovnaký ako odhad u náhodného algoritmu. Celková časová zložitosť je $O(t * (k * n^2 + n * k * \log k + n * k + n \log n + (n - k) * n) + n * k + k \log k + (k - n) * k + p^2 + l))$, kde t je počet vykonaných cyklov.

3.4 Credit Search algoritmus

Tretí algoritmus, ktorý popíšeme používa techniku z CSP. Táto technika sa nazýva credit search. Pod pojmom credit search môžeme si predstaviť limitované hľadanie. Táto technika sa používa u takých typov problémov, kde prehľadávanie stavového priestoru trvá príliš dlho. V tomto prípade sa limituje počet stavov,

ktorých algoritmus môže vyskúšať. Po využívaní všetkých kreditov algoritmus vráti ten stav, ktorý bol najlepší medzi vyskúšanými stavmi. Výhoda algoritmov, ktoré používajú techniku credit search je, že bežia len obmedzený počet krokov. Prehľadávajú len obmedzený počet susedných i nesusedných stavov iniciálneho stavu. V niektorých prípadoch sú rýchlejšie ako úplné algoritmy. Nevýhoda techniky je, že negarantuje nájdenie riešenia problému a ani dokázanie neexistencie riešenia. Algoritmy používajúce credit search sú neúplné, neprehľadávajú celý stavový priestor.

3.4.1 Pseudokód algoritmu

1. **read**(credit).
2. **While** existuje objednávka, ktorá nie je spracovaná **do**
3. **begin**
4. vypočítaj koľko kreditov je potrebných na prehľadávanie všetkých možných párovaní.
5. **if** potrebné kredity $<$ počet kreditov **then** vyskúšaj všetky možné párovania voľných kamiónov a nespracovaných objednávok a vyber najlepší možný z nich
6. **else** vyber náhodne toľko voľných kamiónov u ktorých môžeme vypočítať všetky možné párovania, vyber najlepšie párovanie a pre ostatné voľné kamióny vyber náhodnú objednávku.
7. čakáme dovtedy, kým nestane niektorý kamión alebo niektoré kamióny voľné.
8. Opakuj krok 4 - 7
9. **end**.

3.4.2 Popis príkazov pseudokódu

V prvom riadku príkaz načíta počet kreditov algoritmu.

Na druhom riadku algoritmus testuje, či existuje ešte objednávka, ktorá nie je úplne spracovaná.

Príkaz v štvrtom riadku vypočíta koľko kreditov potrebujeme na úplné prehľadávanie všetkých možných párovaní súčasne voľných kamiónov a súčasne nespracovaných objednávok. Počet potrebných kreditov sa vždy rovná n^k, kde n je počet nespracovaných objednávok a k je počet voľných kamiónov, pretože dovolíme transportovanie tovaru objednávky s viacerými kamiónmi.

V piatom riadku s algoritmom otestujeme, či máme dostatočný počet kreditov na vygenerovanie všetkých možných párovaní. Ak máme dostatočný počet kreditov vygenerujeme všetky párovania. Každé párovanie ohodnotíme jedným kladným reálnym číslom. Najlepšie bude to párovanie, ktoré má minimálnu hodnotu. Túto hodnotu vypočítame nasledovne. Ak u každého páru môže kamión transportovať nenulové množstvo tovaru potom funkčná hodnota párovania sa rovná nasledujúcej sume $\sum_{i=1}^{k} l_i$, kde l_i reprezentuje vzdialenosť od mesta i-tého kamiónu k sklade objednávky, ktorým je v pári. V prípade, že niektoré kamióny môžu transportovať len tovar veľkosti nula pridáme k predošlej sume ešte p - krát pokutu, kde pokuta je reprezentovaná veľmi veľkým číslom a p reprezentuje počet kamiónov, ktoré by mali transportovať tovar veľkosti nula.

V šiestom riadku máme prípad, keď nemáme dostatočný počet kreditov na prehľadávanie všetkých možných párovaní. V tomto prípade rozdelíme kamióny na dve skupiny. V prvej skupine budú tie kamióny pre ktoré vyskúšame všetky možné párovania. V druhej skupine budú tie kamióny pre ktoré generujeme objednávky náhodne. Z počtu kreditov, počtu voľných kamiónov a počtu nespracovaných objednávok vieme vypočítať, že pre koľko voľných kamiónov môžeme vyskúšať všetky možné párovania, toto číslo uložíme do premennej *t*. Do prvej skupiny vyberieme náhodne t kamiónov zo skupiny voľných kamiónov. Zostávajúce kamióny pridáme do druhej skupiny. Po výpočte najlepšieho párovania v prvej skupine, ak exituje kamión v prvej skupine, ktorý by transportoval tovar veľkosti nula tak ten kamión pridáme do druhej skupiny.

V siedmom riadku algoritmus čaká na uvoľnenie kamiónov. Táto operácia je ekvivalentná k príkazu na riadku šesť a sedem v náhodnom algoritme.

3.4.3 Konečnosť algoritmu a determinizmus algoritmu.

Z popisov príkazov algoritmu vyplýva, že algoritmus vykonáva nedeterministické kroky a toto implikuje, že algoritmus je nedeterministický. Z algoritmu je vidieť, že v každom cyklu, každý voľný kamión transportuje nejaké množstvo tovaru. Algoritmus nevykoná taký cyklus v ktorom by sa nezmenšil súčet množstiev tovarov nespracovaných objednávok. Pretože tento súčet je konečný a v každom cyklu zmenšíme súčet je vidieť, že algoritmus je konečný.

3.4.4 Časová zložitosť algoritmu

Pri určení časovej zložitosti algoritmu musíme počítať časovú zložitosť každého hlavného kroku.

Pri inicializácií algoritmu máme vykonať rovnaké operácie ako vykoná prvý algoritmus. V prvom riadku načítanie hodnoty kreditu trvá $O(1)$ času, ktoré je zanedbateľné. U druhého riadku máme rovnaký horný odhad ako u náhodného algoritmu. V štvrtom riadku výpočet trvá $O(1)$ času ak operáciu umocnenia berieme ako konštantnú operáciu. V piatom riadku v najhoršom prípade využijeme všetky kredity a musíme vytvoriť ešte k párov plánov. Celková časová zložitosť tohto príkazu je $O(c + k)$, kde c je počet kreditov a k je počet voľných kamiónov.
V šiestom riadku vykonáme rovnaký počet krokov ako v piatom riadku algoritmu. Celková časová zložitosť je $O(i + t * (c + k))$, kde i je inicializácia algoritmu, t je počet cyklov, c je počet kreditu a k je počet voľných kamiónov.

3.5 Random chance algoritmus

Posledný algoritmus, ktorý popíšeme patrí do skupiny nedeterministických algoritmov. U tohto algoritmu pre každú nespracovanú objednávku spočítame jej preferovací zoznam. Preferovací zoznam obsahuje všetky kamióny, ktoré sú súčasne voľné. Zostalo nám len sa rozhodnúť ako triediť preferovací zoznam. Pre každý kamión vypočítame koľko času by trvalo kamiónu transportovať celý tovar objednávky k zákazníkovi. Tento čas vypočítame nasledujúcim spôsobom. Vypočítame koľkokrát by mal kamión urobiť otočku, aby transportoval celý tovar objednávky, toto číslo uložíme do premennej t a $t = \left\lceil \frac{a}{c} \right\rceil$, kde a je veľkosť tovaru, ktorú máme transportovať a c je kapacita daného kamiónu. Musíme ešte vypočítať akú dlhú cestu musí absolvovať náš kamión, aby obslúžil objednávku. Vieme, že kamión musí sa dostať zo súčasného mesta do mesta skladu objednávky, túto hodnotu uložíme do premennej i. Z databázy vieme získať aká dlhá je najkratšia cesta medzi skladom tovaru objednávky a zákazníkom objednávky, hodnotu dĺžky najkratšej cesty uložíme do premennej l. Celková vzdialenosť, ktorú musí absolvovať kamión, aby obslúžil objednávku označíme písmenom d, kde $d = i + t * 2 * l$. Celkový čas obsluhy sa rovná č $= d \div r$, kde r je rýchlosť kamióna. Preferovací zoznam nespracovanej objednávky triedime podľa celkového času vzostupne.
Pre každý voľný kamión vypočítame, že na akom mieste sa nachádza v preferovacom zozname danej nespracovanej objednávky. Z týchto miest urobíme štatistiku pre kamión. Túto štatistiku uložíme do poľa. Keď vyberieme pre kamión objednávku, pozrieme sa do poľa štatistiky daného kamióna a vyberieme prvý

nenulový prvok na i-tom mieste. Môžu nastať dve možnosti. Pre kamión je len jedna objednávka, v tomto prípade máme presne danú objednávku, ktorú obslúži kamión. V druhom prípade máme viac objednávok, z ktorých môžeme si vybrať náhodne. V druhom prípade vygenerujeme náhodné číslo, ktoré prekonvertujeme na index objednávok. Vyberieme si takto získanú objednávku a vytvoríme plán na obsluhu.

3.5.1 Pseudokód algoritmu

1. **while** existuje objednávka, ktorá nie je spracovaná **do**
2. **begin**
3. pre všetky nespracované objednávky vypočítaj preferovací zoznam
4. pre všetky kamióny vypočítaj štatistiku
5. Pre každý kamión vyhľadáme zo štatistiky prvý nenulový prvok a vytvoríme plán podľa hodnoty prvku.
6. čakáme dovtedy, kým nestane niektorý kamión alebo niektoré kamióny voľné.
7. Opakuj krok 4 - 7
8. **end**.

3.5.2 Popis príkazov pseudokódu

V prvom riadku algoritmus testuje, či existuje ešte objednávka, ktorá nie je úplne spracovaná . V treťom riadku vytvoríme preferovací zoznam pre každú úplne nespracovanú objednávku takým spôsobom ako je napísané na začiatku podkapitoly. V štvrtom riadku vypočítame štatistiky kamiónov podľa preferovacích zoznamov nespracovaných objednávok. V piatom riadku vytvoríme plány pre kamióny, ktoré sú v súčasnosti voľné. Plán vytvoríme podľa dvoch zmienených prípadov. V šiestom riadku čakáme na voľné kamióny.

3.5.3 Konečnosť a determinizmus algoritmu

Algoritmus je konečný. V každom cyklu algoritmu sa zmenší súčet tovaru objednávok, ktorý musíme transportovať k zákazníkovi. Pretože súčet je konečný aj počet cyklov, ktorý vykoná algoritmus je konečný a toto implikuje, že algoritmus je konečný. Algoritmus je nedeterministický. Je vidieť z toho, že v príkaze ,ktorý sa nachádza v piatom riadku pseudokódu algoritmu sa vykonáva generovanie náhodných plánov.

3.5.4 Časová zložitosť algoritmu

U prvého riadku algoritmu na počet vykonaných cyklov máme rovnaký horný odhad ako u prvého algoritmu. V treťom riadku počítame toľko preferovacích zoznamov

koľko máme nespracovaných objednávok. Vytvoriť preferovací zonam trvá $O(k)$ času, kde k je počet voľných kamiónov. Preferovací zoznam musíme ešte aj triediť , čo trvá $O(k \log k)$ operácií. Celková časová zložitosť príkazu je $O(n * k * \log k)$, kde n je počet nespracovaných objednávok . V štvrtom riadku vypočítame štatistiku pre každý voľný kamión. Výpočet pre kamión trvá $O(k^2)$ operácií a máme k voľných kamiónov, z toho vyplýva, že celková časová zložitosť je $O(k^3)$.
V piatom riadku vytvoríme plán pre všetky voľné kamióny. V najhoršom prípade náhodný index musíme generovať zo všetkých nespracovaných objednávok. Celková časová zložitosť je $O(k * n)$. Operácia v šiestom riadku trvá rovnako dlho ako u náhodného algoritmu. Celková časová zložitosť algoritmu $O(l * (n * k * \log k + k^3 + k * n))$, kde l je počet vykonaných cyklov algoritmu.

Kapitola 4

Porovnanie výsledkov testovania algoritmov

V tejto kapitole porovnávame výsledky testov algoritmov. Pre nás budú zaujímavé len nedeterministické algoritmy, pretože u nich každý beh algoritmu s rovnakými vstupnými parametrami, môže generovať rôzne výsledky. Zo štyroch algoritmov máme len jeden deterministický algoritmus, ktorá je modifikovaná verzia algoritmu pánska(dámska) volenka. Pre ostatné tri algoritmy testovacie dáta sú prístupné na CD. Každý algoritmus sme testovali s tromi rôznymi testami. Pri testoch som vyskúšal rôzne typy grafov (rôznymi dĺžkami ciest). Výsledky testov sú znázornené v nasledujúcich troch tabuľkách(tabuľka č.1, č.2. č.3) podľa druhu použitého rozvrhovacieho algoritmu. Prvý stĺpec obsahuje mená testovaných súborov, duhý stĺpec názov rozvrhovacieho algoritmu, tretí stĺpec obsahuje počet plánov, ktoré vytvoril algoritmus pri daných testoch s uvedenou rýchlosťou simulácie. Posledný stĺpec obsahuje čas behu algoritmu pre daný test.

Meno testovaného súbor:	Meno algoritmu	Počet plánov	Rýchlosť simulácie	Čas behu programu
test1.txt	CreditSearch algoritmus	928	0,5	0 h 2 min 21 s
test1.txt	CreditSearch algoritmus	935	0,6	0 h 2 min 2 s
test1.txt	CreditSearch algoritmus	937	0,7	0h 1min 44s
test1.txt	CreditSearch algoritmus	923	1,35	0h 0min 55s
test1.txt	CreditSearch algoritmus	938	1,85	0h 0min 44s
test1.txt	CreditSearch algoritmus	936	1,1	0h 1 min 8 s
test1.txt	CreditSearch algoritmus	928	0,5	0h 2 min 26 s
test1.txt	CreditSearch algoritmus	921	0,8	0 h 1 min 29 s
test1.txt	CreditSearch algoritmus	934	1,25	0 h 1 min 0 s
test1.txt	CreditSearch algoritmus	937	1,1	0 h 1 min 7 s
test2.txt	CreditSearch algoritmus	931	0,5	0 h 4 min 31 s
test2.txt	CreditSearch algoritmus	930	0,6	0 h 3 min 45 s
test2.txt	CreditSearch algoritmus	930	0,8	0 h 2 min 50 s
test2.txt	CreditSearch algoritmus	934	1,35	0 h 1 min 43 s
test2.txt	CreditSearch algoritmus	930	1,85	0 h 1 min 16 s
test2.txt	CreditSearch algoritmus	927	2,25	0 h 1 min 3 s
test2.txt	CreditSearch algoritmus	931	0,5	0 h 4 min 29 s
test2.txt	CreditSearch algoritmus	932	1	0 h 2 min 16 s
test2.txt	CreditSearch algoritmus	934	1.35	0 h 1 min 43 s
test2.txt	CreditSearch algoritmus	931	1.75	0 h 1 min 21 s
test3.txt	CreditSearch algoritmus	1011	0,5	0 h 2 min 20 s
test3.txt	CreditSearch algoritmus	1007	0,6	0 h 1 min 57 s
test3.txt	CreditSearch algoritmus	1020	0,6	0 h 1 min 58 s
test3.txt	CreditSearch algoritmus	1012	1,1	0 h 1 min 5 s
test3.txt	CreditSearch algoritmus	1000	1,2	0 h 1 min 1 s
test3.txt	CreditSearch algoritmus	1043	2,7	0 h 0 min 29 s
test3.txt	CreditSearch algoritmus	1015	0,5	0 h 2 min 22 s
test3.txt	CreditSearch algoritmus	1000	1,2	0 h 1 min 2 s
test3.txt	CreditSearch algoritmus	1023	0,95	0 h 1 min 18 s
test3.txt	CreditSearch algoritmus	1018	0,4	0 h 2 min 56 s

Tabuľka č.1: Výsledky testov s credit search algoritmom

Meno testovaného súbora	Meno algoritmu	Počet plánov	Rýchlosť simulácie	Čas behu programu
test1.txt	RandomChance algoritmus	929	0,5	0 h 2 min 11 s
test1.txt	RandomChance algoritmus	934	0,6	0 h 1 min 52 s
test1.txt	RandomChance algoritmus	930	0,6	0 h 1 min 51 s
test1.txt	RandomChance algoritmus	925	1,2	0 h 0 min 57 s
test1.txt	RandomChance algoritmus	934	1,25	0 h 0 min 54 s
test1.txt	RandomChance algoritmus	932	1,1	0 h 1 min 3 s
test1.txt	RandomChance algoritmus	929	0.5	0 h 2 min 11 s
test1.txt	RandomChance algoritmus	936	0,65	0 h 1 min 43 s
test1.txt	RandomChance algoritmus	933	1,25	0 h 0 min 54 s
test1.txt	RandomChance algoritmus	913	2.05	0 h 0 min 35 s
test2.txt	RandomChance algoritmus	931	0,5	0 h 4 min 27 s
test2.txt	RandomChance algoritmus	934	0,6	0 h 3 min 42 s
test2.txt	RandomChance algoritmus	936	0,6	0 h 3 min 42 s
test2.txt	RandomChance algoritmus	931	1,2	0 h 1 min 53 s
test2.txt	RandomChance algoritmus	936	1,1	0 h 2 min 3 s
test2.txt	RandomChance algoritmus	932	2,4	0 h 1 min 0 s
test2.txt	RandomChance algoritmus	934	0.5	0 h 4 min 24 s
test2.txt	RandomChance algoritmus	932	0.8	0 h 2 min 46 s
test2.txt	RandomChance algoritmus	937	1.1	0 h 2 min 2 s
test2.txt	RandomChance algoritmus	926	2,75	0 h 0 min 52 s
test3.txt	RandomChance algoritmus	1011	0,5	0 h 2 min 20 s
test3.txt	RandomChance algoritmus	1007	0,6	0 h 1 min 57 s
test3.txt	RandomChance algoritmus	1020	0,6	0 h 1 min 58 s
test3.txt	RandomChance algoritmus	1012	1,1	0 h 1 min 5 s
test3.txt	RandomChance algoritmus	1000	1,2	0 h 1 min 1 s
test3.txt	RandomChance algoritmus	1043	2,7	0 h 0 min 29 s
test3.txt	RandomChance algoritmus	1016	0,5	0 h 2 min 19 s
test3.txt	RandomChance algoritmus	1008	0,7	0 h 1 min 41 s
test3.txt	RandomChance algoritmus	998	1.25	0 h 0 min 59 s
test3.txt	RandomChance algoritmus	973	2.4	0 h 0 min 32 s

Tabuľka č.2: Výsledky testov s RandomChance algoritmom

Meno testovaného súbora	Meno algoritmu	Počet plánov	Rýchlosť simulácie	Čas beh programu
test1.txt	Randomizer algoritmus	927	0,5	0 h 2 min 11 s
test1.txt	Randomizer algoritmus	925	0,5	0 h 2 min 1 s
test1.txt	Randomizer algoritmus	925	0,5	0 h 2 min 10 s
test1.txt	Randomizer algoritmus	923	1,2	0 h 0 min 58 s
test1.txt	Randomizer algoritmus	923	1,2	0 h 0 min 57 s
test1.txt	Randomizer algoritmus	931	1,2	0 h 1 min 0 s
test1.txt	Randomizer algoritmus	925	0,5	0 h 2 min 10 s
test1.txt	Randomizer algoritmus	935	0,2	0 h 5 min 26 s
test1.txt	Randomizer algoritmus	929	1,45	0 h 0 min 50 s
test1.txt	Randomizer algoritmus	937	5,5	0 h 4 min 26 s
test2.txt	Randomizer algoritmus	929	0,5	0 h 4 min 28 s
test2.txt	Randomizer algoritmus	929	0,5	0 h 4 min 33 s
test2.txt	Randomizer algoritmus	936	0,5	0 h 1 min 53 s
test2.txt	Randomizer algoritmus	929	1,2	0 h 1 min 53 s
test2.txt	Randomizer algoritmus	940	1,2	0 h 1 min 55 s
test2.txt	Randomizer algoritmus	929	1,2	0 h 1 min 53 s
test2.txt	Randomizer algoritmus	929	0,5	0 h 4 min 26 s
test2.txt	Randomizer algoritmus	937	0,85	0 h 2 min 38 s
test2.txt	Randomizer algoritmus	932	1.3	0 h 1 min 46 s
test2.txt	Randomizer algoritmus	930	1,6	0 h 1 min 27 s
test3.txt	Randomizer algoritmus	1010	0,5	0 h 2 min 20 s
test3.txt	Randomizer algoritmus	1010	0,5	0 h 2 min 20 s
test3.txt	Randomizer algoritmus	1020	0,6	0 h 1 min 58 s
test3.txt	Randomizer algoritmus	1002	1,2	0 h 1 min 1 s
test3.txt	Randomizer algoritmus	1012	1,1	0 h 1 min 5 s
test3.txt	Randomizer algoritmus	991	2,35	0 h 0 min 33 s
test3.txt	Randomizer algoritmus	1010	0,5	0 h 2 min 19 s
test3.txt	Randomizer algoritmus	1017	0.95	0 h 1 min 17 s
test3.txt	Randomizer algoritmus	1008	1,5	0 h 0 min 50 s
test3.txt	Randomizer algoritmus	972	9,95	0 h 0 min 11 s

Tabuľka č.3: Výsledky testov s náhodným(Randomizer) algoritmom

Výsledky testov sme porovnávali Wilcoxonovým testom. Naša hypotéza je, že algoritmy sú porovnateľné z hľadiska výkonu. Wilcoxonový test môžeme nájsť v literatúre[8]. Porovnávali sme Credit Search algoritmus a Second Chance algoritmus. Wilcoxonový test sa skladá z účastníkov. Účastníci v našom prípade budú počty plánov z prvého a druhého testu jednotlivých algoritmov. Ďalej vypočítame po riadkoch rozdiel počtu plánov a absolútne hodnoty rozdielov. Pre každý absolútny rozdiel vypočítame jeho hodnosť. Tento výpočet vykonáme nasledovaným spôsobom. Počítame len nenulové rozdiely, nulové vynecháme.
Ak dva alebo viac rozdielov majú rovnaký absolútny rozdiel (rank), tak každému absolútnemu rozdielu priradíme voľné poradie nasledujúce po sebe také , ktoré by získali keby neboli rovnaké. Poradie sčítame a získanú sumu vydelíme počtom rovnakých hodnôt. Výsledky sú ukázané v nasledovnej tabuľke (tabuľka č.4).

Poradie	CreditSearch algoritmus	SecondChance algoritmus	Rozdiel	Abs. Rozdiel	Zoradený rozdiel
1	928	929	-1	1	2,5
2	935	934	1	1	2,5
3	937	930	7	7	16
4	923	925	-2	2	5
5	938	934	4	4	10
6	936	932	4	4	10
7	928	929	-1	1	2,5
8	921	936	-15	15	17
9	934	933	1	1	2,5
10	937	913	24	24	18
11	931	931	0	0	vynecháme
12	930	934	-4	4	10
13	930	936	-6	6	14,5
14	934	931	3	3	7
15	930	936	-6	6	14,5
16	927	932	-5	5	12,5
17	931	934	-3	3	7
18	932	932	0	0	vynecháme
19	934	937	-3	3	7
20	931	926	5	5	12,5

Tabuľka č.4: Kritické hodnoty Wilcoxonového testu

Spočítame sumu zoradených rozdielov u kladných rozdielov. Túto sumu označíme *p*, $p = 2{,}5 + 16 + 10 + 10 + 18 + 7 + 12{,}5 = 76$. Spočítame sumu aj pre záporne rozdiely, túto sumu označíme *n*, $n = 2{,}5 + 5 + 2{,}5 + 17 + 10 + 14{,}5 + 14{,}5 + 12{,}5 + 7 + 7 = 92{,}5$. Vyberieme menšie číslo z *p* a *n* a označíme ho ako *m*. Spočítame koľko je nenulových rozdielov. V našom prípade je osemnásť nenulových rozdielov. Vyhľadáme z tabuľky Wilcoxonovej kritické hodnoty [http://facultyweb.berry.edu/vbissonnette/tables/wilcox_t.pdf]. Pre nás táto hodnota sa rovná štyridsať. Naše minimum je väčsie ako stanovené minimum, čo znamená že naša hypotéza je platná. Takýmto spôsobom môžeme porovnať všetky dvojice algoritmov z našich implementovaných algoritmov. Platnosť hypotézy doporučujeme overiť s väčším počtom testov.

Kapitola 5
Dokumentácia programov

Moja bakalárska práca sa skladá z dvoch menších programov. Prvý program som nazval Editor a druhý program sa nazýva Simulátor. V prvom programe užívateľ má možnosť vytvoriť vstupné dáta pre Simulátor. Ďalšou funkcionalitou Editoru je možnosť editovať už existujúce testovacie dáta. V druhom programe užívateľ načíta testovacie dáta z textového súboru, vyberie rozvrhovací algoritmus, pridá algoritmus k programu a spustí simuláciu.
Obidva programy som vyvíjal v programovacom jazyku Java SE. Tento jazyk som si vybral preto, že syntax jazyka sa dá veľmi jednoducho naučiť. Ďalšia výhoda jazyka Java je, že má veľmi veľa dopredu naprogramovaných a dobre otestovaných modulov, ktorý programátor môže používať. Hlavným dôvodom použitia jazyka Java je, že je multiplatformový. Kód stačí napísať len raz a rovnaký kód dokáže bežať na rôznych počítačových platformách. Nevýhoda jazyka je, že prekladaný program bežiaci vo virtuálnom prostredí vykonáva inštrukcie oveľa pomalšie ako programy napísané v takom jazyku, ktorý prekladá zdrojový kód na strojový kód.
V tejto kapitole popíšeme užívateľskú a programátorskú dokumentáciu programu Editor a Simulátor. V užívateľskej dokumentácií popíšeme aké požiadavky potrebujú programy na systéme Windows a na systéme Linux, aby sme ich mohli spustiť. Ďalej popíšeme ako sa používajú jednotlivé programy. V programátorskej dokumentácií popíšeme hlavné moduly programov, najdôležitejšie dátové štruktúry a používané cudzie knižnice.

5.1 Užívateľská dokumentácia programu Editor

Na začiatku tejto podkapitoly popíšeme systémové požiadavky programu Editor. Ďalej popíšeme ako nastaviť systém Windows a systém Linux, aby sme mohli spustiť program Editor a popíšeme presne ako sa ovláda program Editor.

5.1.1 Systémové požiadavky a spustenie programu Editor

Program Editor som vyvíjal v Jave vo verzií 1.7. Aby sme mohli spustiť program na našom operačnom systéme ja nutnou podmienkou mať Java JRE 1.7 nainštalované na systéme.

5.1.2 Nastavenie systému Windows a spustenie programu Editor

Keď použijeme systém Windows môžeme ľahko otestovať či máme na našom systéme Javu. S myšou klikneme na štart menu v ľavom dolnom rohu obrazovky a zapíšeme dole príkaz **cmd** a stlačíme kláves **Enter**. Po tejto operácií sa nám otvorí okno príkazového riadku do ktorého zapíšeme príkaz **java –version**. Pokiaľ nám príkazový riadok vráti nasledujúci text „'java' is not recognized as an internal or external command, operable program or batch file“ nemáme na systéme nainštalované funkčné JRE. Inštalátor na JRE môžeme stiahnuť z webovej stránky ***http://www.oracle.com/technetwork/java/javase/downloads/jre7-downloads-1880261.html*** **.** Po inštalácií JRE už môžeme spustiť náš program z príkazového riadku. Na našom systéme musíme nájsť adresár projektu. V adresári **Editor** sa nachádza podadresár **dist** a v tomto adresári je prítomný súbor **Editor.jar.** Tento súbor môžeme spustiť z príkazového riadku pomocou príkazu **java –jar Editor.jar**.

5.1.3 Nastavenie systému Ubuntu a spustenie programu Editor

Môj program som testoval na systéme Ubuntu 14.04, ktorý je súčasná najnovšia verzia systému Ubuntu, ktorý patrí do rodiny Linux systémov a je jedným najpopulárnejším systémom rodiny Linux.

Ak chceme otestovať verziu Javy jednoducho otvoríme terminál a zapíšeme do terminálu príkaz java –version rovnako ako u systému Windows. V prípade, že Java nie je prítomná na systéme, môžeme nainštalovať JRE pomocou príkazu **sudo apt-get install default-jre.** Po inštalácií Javy konečne môžeme spustiť náš program. Vyhľadáme v termináli adresár kde sa nachádza súbor **Editor.jar** a spustíme program príkazom **java –jar Editor.jar**, ktorý je rovnaký príkazu používaného na systéme Windows.

5.1.4 Užívateľská dokumentácia

Program Editor okrem dvoch hlavných funkcionalít, ktoré sme zmienili má aj iné funkcionality. Funkcionalitu programu rozdelíme do troch veľkých skupín. Do prvej skupiny patria také funkcie, ktoré manipulujú so súbormi, načítajú dáta zo súboru alebo uložia dáta z databázy do súboru. Do druhej skupiny patria také funkcie, ktoré menia chovanie programu a do tretej skupiny patria také funkcie, ktoré menia vlastnosti dát v databáze.

5.1.4.1 Položky Menu programu Editor

Funkcie prvej skupiny sa nachádzajú v menu časti programu. Dáta databázy obsahujú štyri hlavné objekty. Medzi objekty patria mesto, cesta, kamión a objednávka. V menu každý z týchto štyroch objektov má vlastnú položku na načítanie z textového súboru (**Open a city file, Open a road file, Open a truck file, Open a package file**).

Položka menu **Open a city file** načíta mestá z textového súboru. V textovom súbore môžu byť len mestá. Pri načítaní otestuje, či všetky načítané parametre mesta sú validné. Pri nevalidných dátach hlási koľký objekt je chybný v súbore. Pred uložením mesta do databázy otestuje či náhodou dané mesto už nie je prítomné v databáze. Tieto testy sú vykonané u každého načítaného objektu.

Položka menu **Open a road file** načíta všetky cesty z textového súboru a otestuje aj všetky parametre cesty. U objektu cesty nie je dovolené, aby cesta sa začínala a skončila v rovnakom meste a dĺžka cesty môže byť len kladné reálne číslo. Pred uložením do databázy program otestuje či exitujú mestá, ktoré cesta spojí.

Položka menu **Open a truck file** načíta všetky kamióny z textového súboru. U kamiónov nie je dovolené aby rýchlosť, kapacita a cena za jednotku vzdialenosti kamióna bola nulová alebo záporné reálne číslo.

Položka menu **Open package file** načíta všetky objednávky. U objednávky nie je dovolené aby sklad tovaru objednávky a zákazník objednávky sa nachádzali v rovnakom meste. Množstvo tovaru môže byť len kladné reálne číslo.

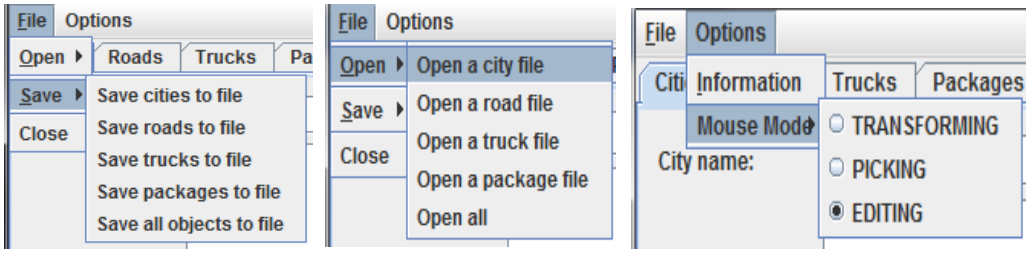
Okrem týchto štyroch existuje aj piata položka menu **Open all**, ktorá načíta objekty z takého textového súboru, ktorý obsahuje mestá, cesty, kamióny a objednávky.

Na uloženie dát do textového súboru máme nasledujúce položky menu (**Save cities to file, Save roads to file, Save trucks to file, Save packages to file, Save all objects to file**). Položka menu **Save cities to file** uloží všetky mestá z databázy do vybraného textového súboru. Položka menu **Save roads to file** uloží všetky cesty z databázy do vybraného textového súboru. Položka menu **Save trucks to file** uloží všetky kamióny z databázy do vybraného textového súboru. Položka menu **Save packages to file** uloží všetky objednávky z databázy do vybraného textového súboru. Posledná položka menu na uloženie dát do vybraného textového súboru je položka menu **Save all objects to file**. Táto položka menu uloží všetky mestá, cesty, kamióny a objednávky do spoločného súboru.

Okrem týchto položiek menu sú ďalšie tri položky menu. Prvá plozka je položka menu **Close**, ktorá sa spýta užívateľa, či chce užívateľ ukončiť aplikáciu. Ak užívateľ na dialógu klikne na tlačidlo **Yes** potom aplikácia položí užívateľovi ďalšiu otázku. Aplikácia sa spýta užívateľa či chce uložiť všetky objekty databázy do textového súboru.

Druhá položka menu Information, ktorá vypíše v dialógu informácie o programe Editor. Posledná položka menu je **Mouse Mode**, ktorá patrí do druhej skupiny funkcionality. Táto položka zmení chovanie myši u grafického tabelátora.

Nasledujúci obrázok č.2 ilustruje všetky položky menu programu Editor.



Obrázok č.2.: Položky menu programu Editor

5.1.4.2 Tabelátory programu Editor

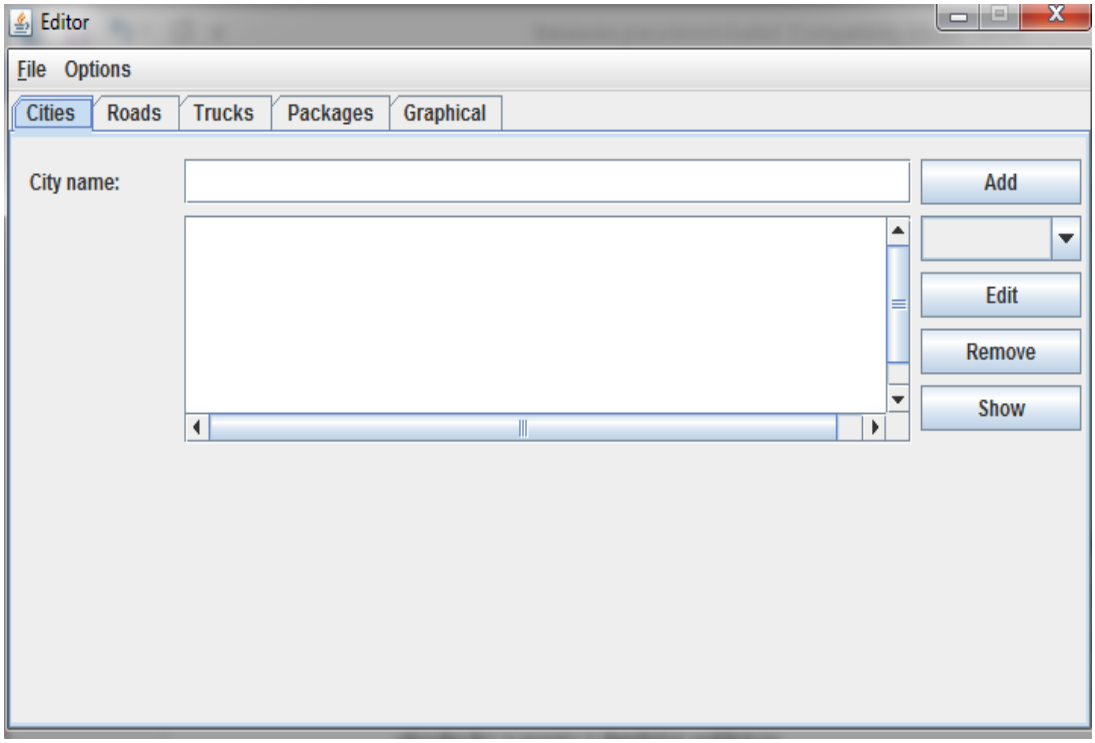
Tabelátory programu Editor patria do tretej skupiny funkcionality. Máme päť rôznych tabelátorov (**City tab, Road tab, Truck tab, Package tab, Graphical tab**). V týchto tabelátoroch môžeme editovať vlastnosti objektov, odstrániť objekty a pridať nové objekty.

Prvý tabelátor, ktorý popíšeme je **City tab** (obr.č.3). V prvom tabelátori môžeme pridať nové mestá do databázy aplikácie pomocou tlačidlo **Add**. U nového pridaného mesta jediná vlastnosť, ktorú môžeme zadať je meno mesta. Meno mesta musí dodržať nasledujúci formát. Meno mesta sa skladá z jedného slova, začína sa s veľkým písmenom, ktoré nasleduje ľubovoľný počet malých písmen a slovo môže končiť na ľubovoľný počet číslic. Medzi ďalšími funkcionalitami tabelátora patrí editovanie mena mesta, odstránenie mesta z databázy a zobrazovanie informácií o vybranom meste.

Keď chceme zobraziť informáciu o meste, vyberieme dané mesto z poľa zoznamov a klikneme na tlačidlo **Show**. Po stlačení tlačidlo **Show** opýtaná informácia sa zobrazí v textovej oblasti tabelátora. Medzi zobrazenými informáciami je unikátne identifikačné číslo mesta, meno mesta, a súradnice mesta, kde sa nachádza na grafickom tabelátori.

Pri editovaní mena mesta, zapíšeme nové meno mesta do textového poľa, vyberieme mesto z poľa zoznamov a klikneme na tlačidlo **Edit**.

Posledná funkcionalita tabelátora je odstránenie mesta z databázy aplikácie. Po kliknutí na tlačidlo **Remove**, program vypočíta koľko ciest, kamiónov a objednávok sa stalo invalidným. Po tomto výpočte program sa spýta užívateľa či určite chce odstrániť mesto. Pri kladnej odpovedi odstráni všetky invalidné cesty, kamióny, objednávky a mesto z databázy aplikácie a aktualizuje grafickú reprezentáciu grafu ciest a miest.



Obrázok č.3.: Tabelátor mesta - City tab

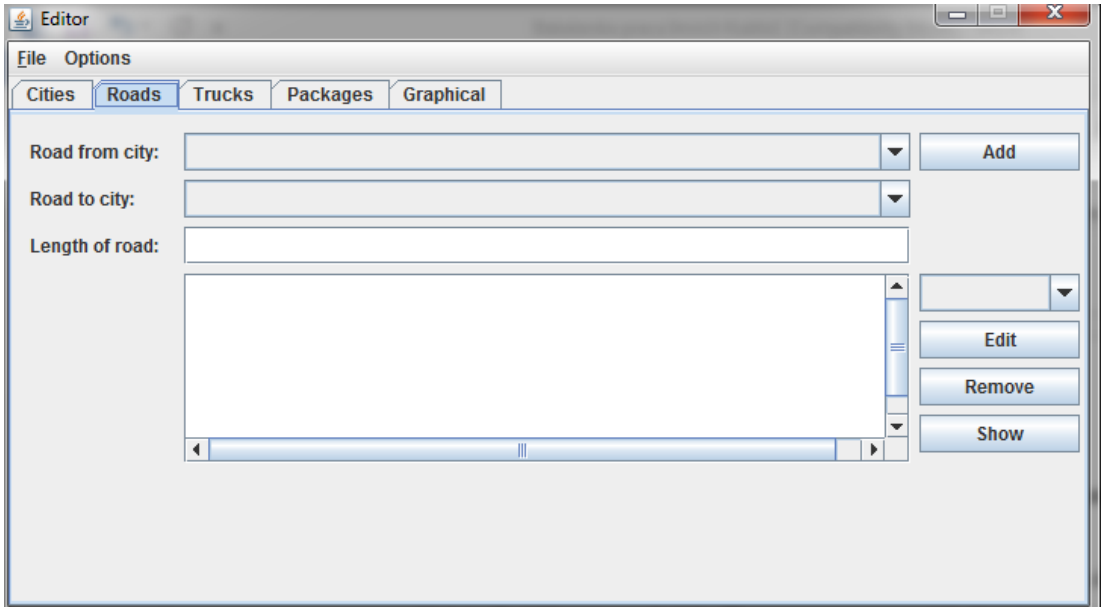
Druhý tabelátor, ktorý popíšeme je **Road tab** (obr.č.4). V druhom tabelátori môžeme pridať nové cesty do databázy aplikácie pomocou tlačidlo **Add**. Pred pridaním novej cesty môžeme nastaviť mesto odkiaľ a kam vedie cesta, dĺžku neorientovanej cesty. Pri pridaní novej neorientovanej cesty aplikácia otestuje, či cesta s takýmito vlastnosťami už nie je prítomná v databáze, ak je neprítomná pridá do databázy dve orientované cesty, ktoré reprezentujú danú neorientovanú cestu.

Keď chceme zobraziť informáciu o ceste, vyberieme danú cestu z poľa zoznamov a klikneme na tlačidla **Show**. Po stlačení tlačidla **Show** opýtaná informácia sa zobrazí v textovej oblasti tabelátora. Medzi zobrazenými informáciami je unikátne

identifikačné číslo cesty, identifikačné číslo mesta odkiaľ vedie cesta, identifikačné číslo mesta kam vedie cesta a dĺžka cesty.

Pri editovaní cesty, nastavíme nové mestá odkiaľ a kam vedie cesta a nastavíme novú dĺžku cesty. Po kliknutí na tlačidlo **Edit** program otestuje nové vlastnosti neorientovanej cesty a keď všetko je validné nastaví program nové vlastnosti orientovaných ciest, ktoré reprezentujú danú neorientovanú cestu.

Posledná funkcionalita tabelátora je odstránenie neorientovanej cesty z databázy aplikácie. Po kliknutí na tlačidlo **Remove**, program vyhľadá orientované cesty, ktoré reprezentujú danú orientovanú cestu a odstráni ich z databázy aplikácie.

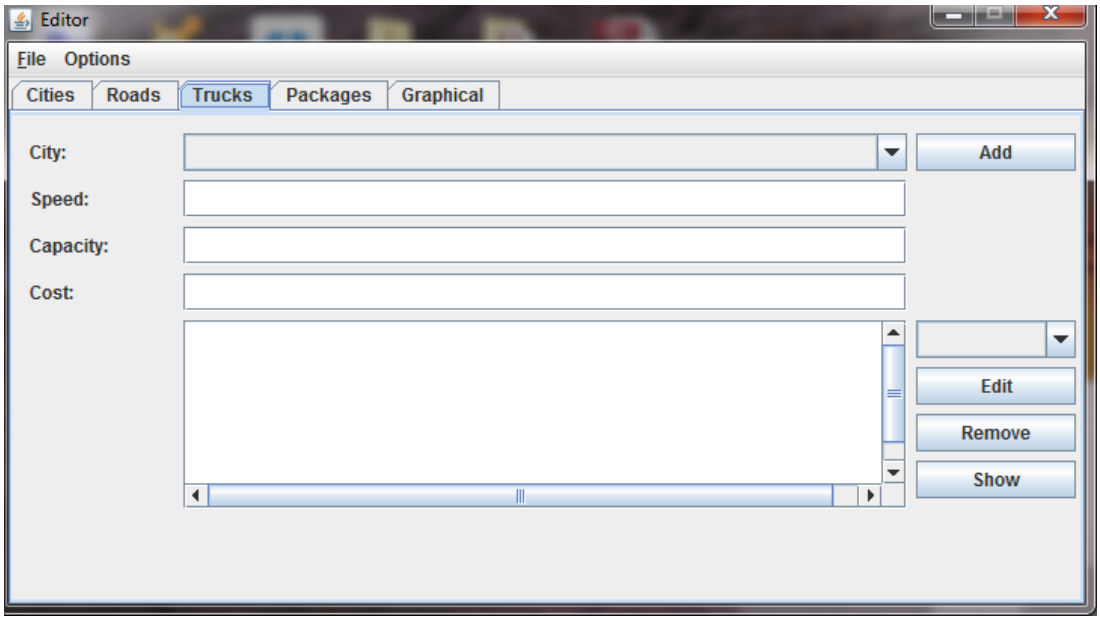


Obrázok č.4.: Tabelátor cesty – Road tab

Tretí tabelátor, ktorý popíšeme je **Truck tab** (obr.č.5). V treťom tabelátori môžeme pridať nové kamióny do databázy aplikácie pomocou tlačidlo **Add**. Pred pridaním nového kamiónu môžeme nastaviť mesto (odkiaľ štartuje kamión svoju prácu), kapacitu, rýchlosť a cenu za jednotku vzdialenosti kamiónu. Pri pridaní nového kamióna aplikácia otestuje, či existuje mesto odkiaľ štartuje kamión svoju prácu a otestuje ešte či hodnota rýchlosti, kapacity a ceny za jednotku vzdialenosti kamiónu je reprezentovaná kladným reálnym číslom.

Keď chceme zobraziť informáciu o kamióne, vyberieme dané mesto z poľa zoznamov a klikneme na tlačidlo **Show**. Po stlačení tlačidla **Show** opýtaná informácia sa zobrazí v textovej oblasti tabelátora. Medzi zobrazenými informáciami je unikátne identifikačné číslo kamiónu, identifikačné číslo mesta, kde sa nachádza kamión, kapacita, rýchlosť a cena kamiónu.

Pri editovaní kamiónu, nastavíme nové mesto odkiaľ štartuje kamión svoju prácu, novú rýchlosť, kapacitu a cenu za jednotku vzdialenosti kamióna.
Po kliknutí na tlačidlo **Edit**, program otestuje nové vlastnosti kamiónu a keď všetko je validné nastaví program nové vlastnosti kamiónu.
Posledná funkcionalita tabelátora je odstránenie kamióna z databázy aplikácie. Po kliknutí na tlačidlo **Remove**, program vyhľadá vybraný kamión z databázy aplikácie a odstráni ju.

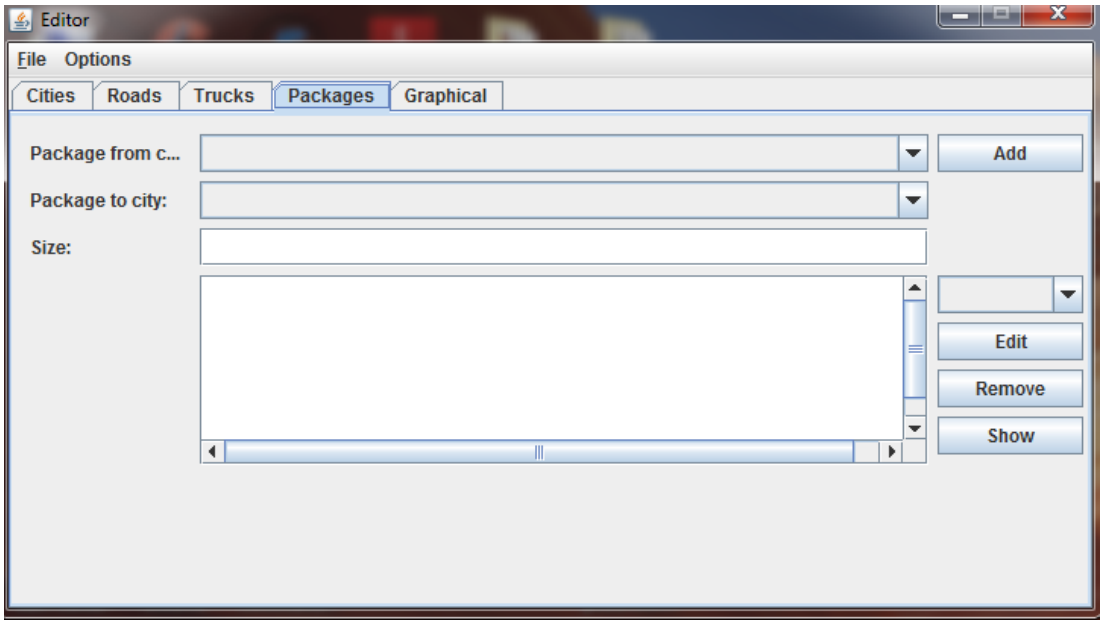


Obrázok č.5.:Tabelátor kamiónu - truck tub

Štvrtý tabelátor, ktorý popíšeme je **Package tab**. V štvrtom tabelátori môžeme pridať nové objednávky do databázy aplikácie pomocou tlačidla **Add**. Pred pridaním novej objednávky môžeme nastaviť mesto skladu tovaru objednávky, mesto zákazníka a množstvo tovaru objednávky, ktoré musíme transportovať. Pri pridaní novej objednávky aplikácia otestuje, či mesto skladu tovaru objednávky je odlišné od mesta zákazníka a otestuje, či množstvo je zadané s kladným reálnym číslom.
Keď chceme zobraziť informáciu o objednávke, vyberieme danú objednávku z poľa zoznamov a klikneme na tlačidlo **Show**. Po stlačení tlačidla **Show** opýtaná informácia sa zobrazí v textovej oblasti tabelátora. Medzi zobrazenými informáciami je unikátne identifikačné číslo objednávky, identifikačné číslo mesta kde sa nachádza sklad tovaru objednávky, identifikačné číslo mesta, kde sa nachádza zákazník a množstvo tovaru objednávky.
Pri editovaní objednávky, nastavíme nové mesto, kde sa nachádza sklad tovaru objednávky, nové mesto zákazníka a nové množstvo tovaru objednávky. Po kliknutí

na tlačidlo **Edit** program otestuje nové vlastnosti objednávky a keď všetko je validné nastaví program nové vlastnosti objednávky.

Posledná funkcionalita tabelátora je odstránenie objednávky z databázy aplikácie. Po kliknutí na tlačidlo **Remove**, program vyhľadá objednávku, ktorú sme vybrali a odstráni ju z databázy aplikácie.



Obrázok č.6.: Tabelátor objednávky - Package tab

Posledný tabelátor, ktorý popíšeme je **Graphical tab** (obr.č.7)**.** Tabelátor **Graphical tab** vizualizuje sieť ciest a miest na tabelátori. Pri práci na tabelátori myš má tri možné módy (**Editing, Picking, Transforming**). Pri každom móde myši je možné blížiť kameru tabulátora alebo oddialiť kolieskom myši.

Pri možnosti Editing, keď klikneme s ľavým tlačidlom myši na také miesto tabelátora, kde sa nenachádza žiadne mesto potom program vytvorí nový vrchol grafu a pridá do databázy aplikácie nové mesto. Meno nového mesta vytvorí tak, že k reťazci „City“ sa pridá ešte identifikačné číslo nového mesta.

Keď klikneme s ľavým tlačidlom myši na také miesto v tabelátori, kde je už prítomný vrchol a ťaháme myš dovtedy, kým nenarazíme na ďalší vrchol a pri tejto operácií stále držíme ľavé tlačidlo myši, potom program vytvorí novú hranu grafu, ak hrana ešte neexistuje v grafe. Pri pridaní novej hrany program vykreslí neorientovanú hranu na **Graphical tab** a pridá dve orientované cesty do databázy aplikácie. Nové pridané orientované cesty majú automaticky dĺžku sto jednotiek.

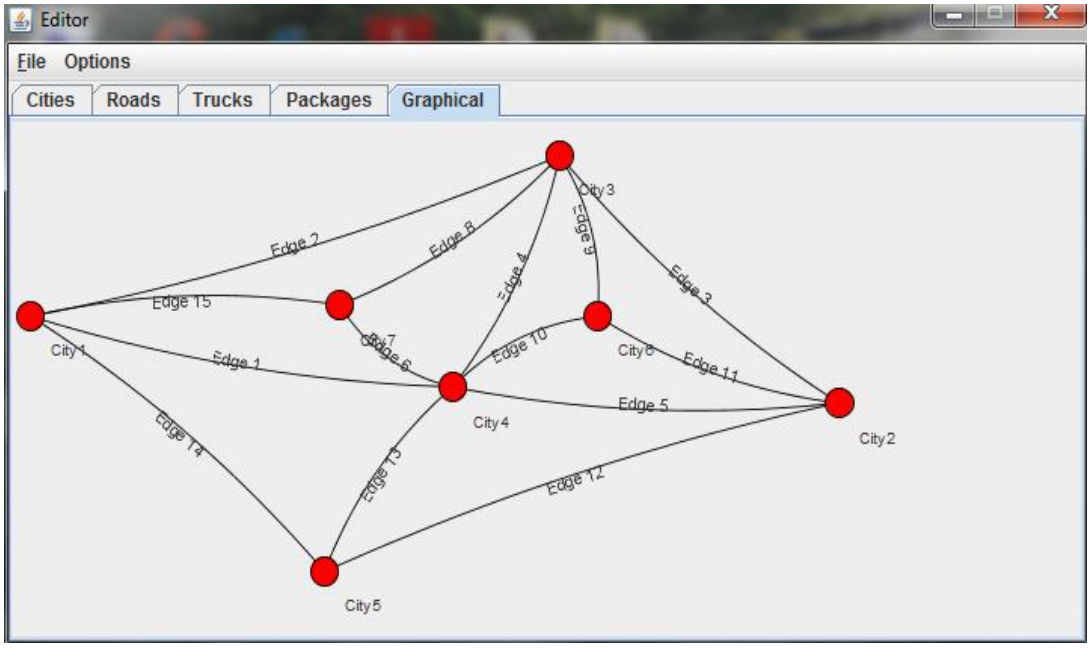
Kliknutím na vrchol grafu s pravým tlačidlom myši sa zobrazí menu vrcholu. Menu vrcholu obsahuje informácie o danom meste a možnosť odstrániť vrchol z grafu.

Keď klikneme na odstránenie vrcholu, program sa nás spýta, či určite chceme odstrániť vrchol. Ako odpovieme na danú otázku kladnou odpoveďou, program odstráni

všetky neorientované hrany z grafu, ktoré sú spojené s vrcholom a tiež vrchol z grafickej reprezentácie sieti ciest a miest. Pri odstránení vrcholu z grafickej reprezentácie program odstráni orientované cesty a objednávky z databázy, ktoré sú v spojení s mestom, kamióny ktoré sa nachádzajú v meste a dané mesto z databázy. Kliknutím na hranu sa objaví v **Graphical Tab** menu hrany. V menu hrany je informácia o tom, aká dlhá je cesta medzi vrcholmi a tiež máme ešte možnosť odstrániť neorientovanú hranu z grafickej reprezentácie. Odstránenie hrán a vrcholov grafickej reprezentácie je dovolené v každom stave myši.

Keď má myš stav **Picking**, potom môžeme vybrať jeden vrchol na ktorý klikneme alebo niekoľko rôznych vrcholov. Keď držíme ľavé tlačidlo myši a kurzor myši súčasne sa nachádza na vrchole grafu môžeme zmeniť lokalitu vrcholu. Pri zmene lokality vrcholu sa prekreslí grafická reprezentácia grafu. Keď držíme ľavé **ctrl** na klávesnici a klikneme na nejaký vrchol grafu potom kamera grafického tabelátora sa zaostrí na vybraný vrchol.

Pri stave myši **Transforming,** keď držíme ľavé **ctrl** a súčasne držíme ľavé tlačidlo myši potom môžeme roztiahnuť graf podľa súradnice kurzorom myši. Keď držíme ľavý **shift** a ľavé tlačidlo myši, potom môžeme rotovať graf na ľavo alebo na pravo.



Obrázok č.7.: Grafický tabelátor – Graphical tab

5.2 Programátorská dokumentácia programu Editor

Program Editor som vyvíjal v **Java SE** verzií **1.7**. Pre vývoj som použil vývojové prostredie **Netbeans 7.4**. Program sa skladá z jedného veľkého hlavného modulu. Tento modul sa nazýva **Editor Frame**, ktorý sa delí na menšie podmoduly. Medzi podmoduly patrí menu bar Editoru, tabelátor miest, tabelátor ciest, tabelátor kamiónov, tabelátor objednávok a grafický tabelátor. Každý má svoju funkcionalitu, ktorú sme popísali v užívateľskej dokumentácií. Aby tieto moduly mohli spolu komunikovať vytvoril som jeden menší modul, cez ktorý moduly komunikujú, tento modul som nazval **Databáza**. Implementácia modulu sa nachádza v zdrojovom kóde **EditorDatabase.java.** Zdrojové kódy programu som rozdelil do dvoch väčších skupín.

Prvá skupina sa nazýva **Editor data structures** a nachádza sa v balíku (*package)* EditorDataStructures, ktorý môžeme nájsť v adresári **Editor/src.** U každého zdrojového kódu som okomentoval funkcie, že akú operáciu vykonávajú. Z komentára sa dá vygenerovať **Javadoc**.

Druhá skupina sa nazýva **Editor** a nachádza sa tiež v adresári **Editor/src**. V tomto adresári sa nachádza grafika programu a hlavné moduly programu.

5.2.1 Dátové štruktúry programu Editor

Už vieme, že zdrojové kódy sú rozdelené do dvoch skupín. Z prvej skupiny by som chcel zdôrazniť najdôležitejšie triedy (**class**), ktoré tvoria základ celej aplikácie. Medzi tieto triedy patrí trieda **City**, trieda **Road**, trieda **Truck**, trieda **Package,** trieda **CityVertex**,trieda **RoadEdge** a trieda **EditorDatabase**. Implementáciu týchto tried môžeme nájsť v súboroch **City.java, Road.java, Truck.java, Package.java, CityVertex.java, RoadEdge.java** a **EditorDatabase.java**.

Trieda **City** modeluje všetky vlastnosti mesta, **Road** modeluje orientovanú cestu, **Truck** modeluje všetky dôležité vlastnosti kamiónu, ktoré sú používané pri simulácií. Trieda **Package** modeluje vlastnosti objednávky, CityVertex reprezentuje mesto v grafe ako vrchol a RoadEdge reprezentuje neorientovanú cestu. Posledné dve triedy sú požívané pri grafickej reprezentácií miest a ciest. Trieda EditorDatabase obsahuje všetky mestá, cesty, kamióny a objednávky, zaistí rýchly prístup k objektom v databáze. K cestám a mestám v databáze sa dá dostať v $O(1)$

čase a ku kamiónom a objednávkam v $O(\log n)$, kde n je počet kamiónov alebo objednávok.
V druhej skupine tried sú tie triedy, ktoré implementujú grafiku a funkcionalitu aplikácie. Medzi najdôležitejšie triedy patrí trieda EditorFrame, trieda CityTab, trieda **RoadTab**, trieda **TruckTab**, trieda **PackageTab** a trieda **GraphicalTab**. **EditorFrame** je oknom celej aplikácie a dedí vlastnosti triedy JFrame. Každý tabelátor dedí vlastnosti od triedy **JPanel.** Tabelátory sú implementované v java súboroch **CityTab.java, RoadTab.java, TruckTab.java, PackageTab.java** a **GraphicalTab.java**.
Trieda **CityTab** zaistí grafickú reprezentáciu tabelátora mesta a zaistí funkcionalitu tabelátora (pridanie nového mesta, odstránenie mesta, editovanie vlastností mesta a textové zobrazenie informácií o meste).
Trieda **RoadTab** zaistí grafickú reprezentáciu tabelátora cesty a zaistí funkcionalitu tabelátora (pridanie novej cesty, odstránenie cesty, editovanie vlastností cesty a textové zobrazenie informácií o ceste).
Trieda **TruckTab** zaistí grafickú reprezentáciu tabelátora kamióna a zaistí funkcionalitu tabelátora (pridanie nového kamióna, odstránenie kamióna, editovanie vlastností kamióna a textové zobrazenie informácií o kamióne).
Trieda **PackageTab** zaistí grafickú reprezentáciu tabelátora objednávky a zaistí funkcionalitu tabelátora (pridanie novej objednávky odstránenie objednávky, editovanie vlastností objednávok a textové zobrazenie informácií o objednávke).
Trieda GraphicalTab zaistí grafickú reprezentáciu sieti miest a ciest, ktorá je reprezentovaná pomocou grafu. Do funkcionality grafického tabelátora patrí pridanie a odstránenie mesta z grafu a pridanie a odstránenie cesty z grafu. U grafického tabelátora som použil cudziu knižnicu **JUNG verzie 2.0.1.**

5.2.2 Cudzie knižnice

Pri grafickej reprezentácií knižnice som použil knižnicu **JUNG 2.0.1.** (http://jung.sourceforge.net/). Táto knižnica má otvorený voľne editovateľný zdrojový kód. Knižnica ušetrí veľa času programátora a zjednoduší kód aplikácie. Vyskúšal som aj iné knižnice, pre kreslenie grafov, ako je napríklad **Graphstream (**http://graphstream-project.org/doc/Tutorials/Graph-Visualisation_1.1/), ale pre môj účel najviac vyhovovala knižnica **JUNG.** Do projektu sú pridané **jar** súbory knižnice **JUNG** v adresári **Editor/jung2-2_0_1**.

5.3 Užívateľská dokumentácia programu Simulator

Na začiatku tejto podkapitoly popíšeme systémové požiadavky programu Simulator. Ďalej popíšeme ako nastaviť systém Windows a systém Linux, aby sme mohli spustiť program Simulator a popíšeme presne ako sa ovláda program Simulator.

5.3.1 Systémové požiadavky a spustenie programu Simulator

Program Simulator som vyvíjal v **Jave vo verzií 1.7.** Aby sme mohli spustiť program na našom operačnom systéme ja nutnou podmienkou mať Java JRE 1.7 nainštalované na systéme.

5.3.2 Nastavenie systému Windows a spustenie programu Simulator

Keď použijeme systém Windows môžeme ľahko otestovať, či máme na našom systéme Javu. S myšou klikneme na štart menu v ľavom dolnom rohu obrazovky a zapíšeme dole príkaz **cmd** a stlačíme kláves **Enter**. Po tejto operácií sa nám otvorí okno príkazového riadku do ktorého zapíšeme príkaz **java –version**. Pokiaľ nám príkazový riadok vráti nasledujúci text „'java' is not recognized as an internal or external command, operable program or batch file“ nemáme na systéme nainštalované funkčné JRE. Inštalátor na JRE môžeme stiahnuť z webovej stránky ***http://www.oracle.com/technetwork/java/javase/downloads/jre7-downloads-1880261.html*** **.** Po inštalácií JRE už môžeme spustiť náš program z príkazového riadku. Na našom systéme musíme nájsť adresár projektu. V adresári **Simulator** sa nachádza podadresár **dist** a v tomto adresári je prítomný súbor **Simulator.jar.** Tento súbor môžeme spustiť z príkazového riadku pomocou príkazu **java –jar Simulator.jar**. Pred spustením je nutné pridať do adresára **dist** obrázok **truck.png**.

5.3.3 Nastavenie systému Ubuntu a spustenie programu Simulator

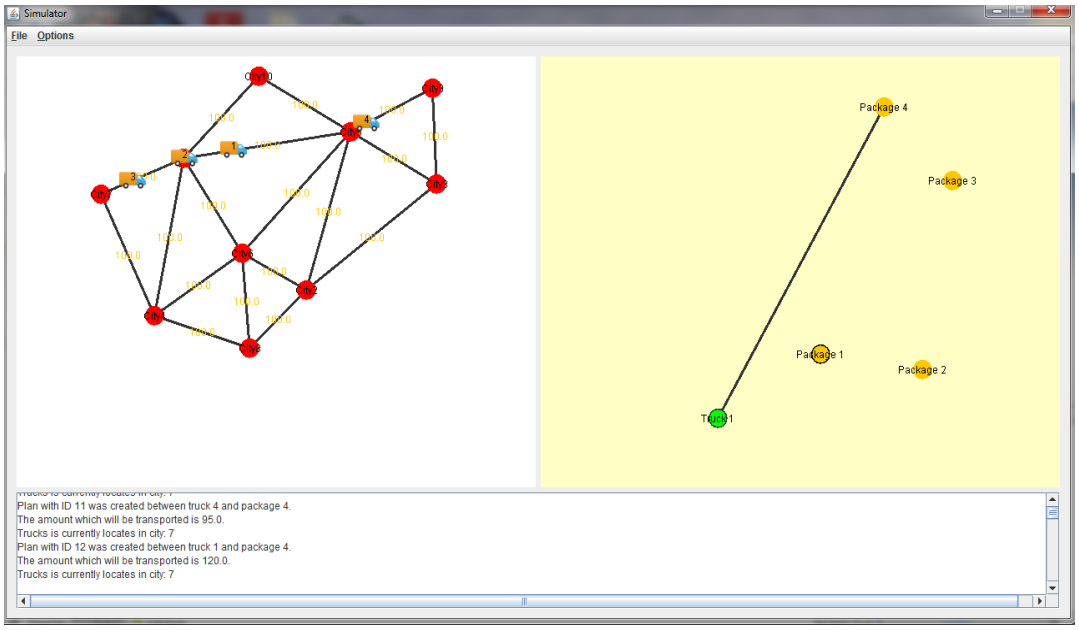
Môj program som testoval na systéme Ubuntu 14.04, ktorý je súčasná najnovšia verzia systému Ubuntu, ktorý patrí do rodiny Linux systémov a je jedným najpopulárnejším systémom rodiny Linux.

Ak chceme otestovať verziu Javy jednoducho otvoríme terminál a zapíšeme do terminálu príkaz java –version rovnako ako u systému Windows. V prípade, že Java nie je prítomná na systéme, môžeme nainštalovať JRE pomocou príkazu **sudo apt-get install default-jre.** Po inštalácií Javy konečne môžeme spustiť náš program.

Vyhľadáme v termináli adresár, kde sa nachádza súbor **Simulator.jar** a spustíme program príkazom **java –jar Simulator.jar**, ktorý je rovnaký príkazu používaného na systéme Windows.

5.3.4 Užívateľská dokumentácia

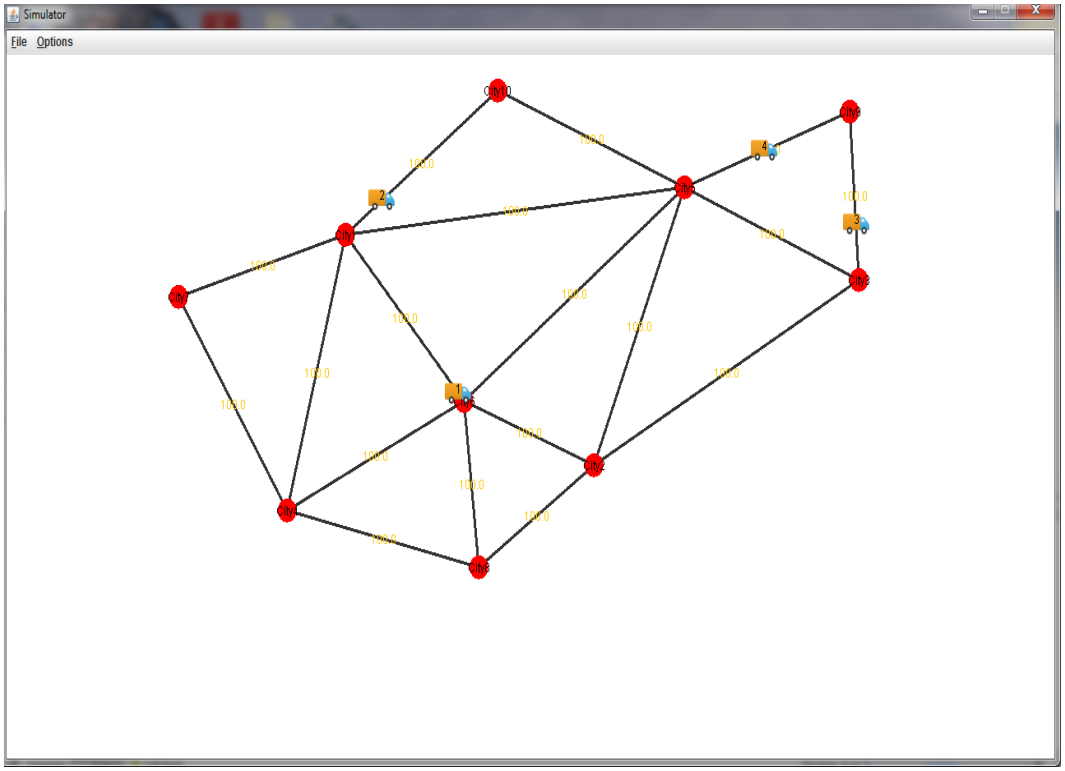
Program Simulator pri spustení sa líši od programu Editor s tým, že máme možnosť zadať ešte dva vstupné parametre. Prvý parameter je konštantný reťazec „**Debug**“, ktorý zmení usporiadanie grafických komponentov. Druhý parameter je prirodzené číslo, ktoré reprezentuje koľko kreditov dovolíme algoritmu **Credit search.** Pri spustení programu v testovacom móde zadáme príkaz v nasledujúcom formáte **java –jar Simulator.java Debug**, ak chceme ešte nastaviť aj kredit algoritmu, zavoláme program s nasledovným spôsobom **java –jar Simulator.java Debug 500**.



Obrázok č.8.: Program je spustený príkazom **java –jar Simulator.jar Debug 100.**

V testovacom móde program vykreslí aj ďalšie dva komponenty. Prvý komponent je testovacie okienko nachádzajúci sa na pravej strane aplikácie. V tomto okienku môžeme sledovať kroky algoritmu. V tomto okne objednávky a kamióny sú prezentované vrcholmi. Vrcholy objednávok majú dve rôzne farby, ktoré implikujú či objednávka je aktívna v súčasnom rozvrhovacom cyklu algoritmu, pokiaľ objednávka má oranžovú farbu, potom objednávka sa zúčastní v súčasnom cyklu rozvrhovania. Keď má objednávka žltú farbu, potom sa nezúčastní v súčasnom cyklu rovrhovania. Rovnako som vytvoril farby pre kamióny, pretože môžeme vytvoriť

algoritmus, ktorý rozdelí súčasne voľné kamióny na aktívne a pasívne kamióny. Aktívne kamióny majú zelenú farbu a pasívne kamióny majú červenú farbu. Okienko v doľnej časti aplikácie je výstup algoritmu, ktorý popíše, ktorý aktívny kamión ktorú objednávka obslúži.



Obrázok.č.9.:Program Simulator spustený s príkazom java –jar Simulator.jar

Keď spustíme simulator bez dopredu zadefinovaných parametrov, potom môžeme len sledovať kamióny ako obsluhujú objednávky.

Celá funkcionalita programu je sústredená do **menu bar** programu.

5.3.4.1 Položky Menu programu Simulator

Funkcionalita programu sa nachádza v menu časti programu. Dáta databázy obsahujú päť hlavných objektov. Medzi objekty patria mesto, cesta, kamión, objednávka a najkratšie cesty v sieti ciest a miest. Všetky tieto objekty sú chránené pred editovaním s viacerými vláknami súčasne. V menu mesto, cesta, kamióny a objednávky majú svoju vlastnú položku na načítanie z textového súboru (**Open a city file, Open a road file, Open a truck file, Open a package file**).

Položka menu **Open a city file** načíta mestá z textového súboru. V textovom súbore môžu byť len mestá. Pri načítaní otestuje, či všetky načítané parametre mesta sú validné. Pri nevalidných dátach hlási koľký objekt je chybný v súbore. Pred

uložením mesta do databázy otestuje či náhodou dané mesto už nie je prítomné v databáze. Tieto testy sú vykonané u každého načítaného objektu.
Položka menu **Open a road file** načíta všetky cesty z textového súboru a otestuje aj všetky parametre cesty. U objektu cesty nie je dovolené, aby cesta sa začínala a skončila v rovnakom meste a dĺžka cesty môže byť len kladné reálne číslo. Pred uložením do databázy program otestuje či exitujú mestá, ktoré cesta spojí.
Položka menu **Open a truck file** načíta všetky kamióny z textového súboru.
U kamiónov nie je dovolené aby rýchlosť, kapacita a cena za jednotku vzdialenosti kamióna bola nulová alebo záporné reálne číslo.
Položka menu **Open package file** načíta všetky objednávky. U objednávky nie je dovolené, aby sklad tovaru objednávky a zákazník objednávky sa nachádzali v rovnakom meste. Množstvo tovaru môže byť len kladné reálne číslo.
Okrem týchto štyroch existuje aj piata položka menu **Open all**, ktorá načíta objekty z takého textového súboru, ktorý obsahuje mestá, cesty, kamióny a objednávky.
Okrem týchto položiek menu sú ďalšie štyri položky menu. Prvá položka je položka menu **Close**, ktorá sa spýta užívateľa, či chce užívateľ ukončiť aplikáciu. Ak užívateľ na dialógu klikne na tlačidlo **Yes** potom aplikácia sa ukončí. Druhá položka menu je **Simulator information,** ktorá vypíše v dialógu informácie o programe Simulator.
Tretie položka je **Simulation speed**, ktorým môžeme nastaviť rýchlosť simulácie. Dáva možnosť nastaviť rýchlosť aj pred simuláciou a v behom simulácie tiež.
Posledná položka menu je **Start simulation**, ktorou môžeme naštartovať beh simulácie. **Pred štartom simulácie je nutnou podmienkou načítať nejakú sieť ciest a miest, kamióny a objednávky z textového súboru!** Keď nenačítame žiadny textový súbor s dátami, potom pri štartovaní simulácie program nevykoná žiadnu operáciu. Pri štarte simulácie program sa nás spýta, aby sme vybrali algoritmus pre simuláciu. Vyhľadáme algoritmus z balíka Algorithms, ktorý sa nachádza v adresári Simulator\build\classes\Algorithm a klikneme na tlačidlo OK. Po tejto operácií program otestuje vlastnosti algoritmu a keď všetky testy sú v poriadku naštartuje simulácia.

5.4 Programátorská dokumentácia programu Simulator

Program Simulator som vyvíjal v **Java SE** vo verzií **1.7**. Pre vývoj som použil vývojové prostredie **Netbeans 7.4**. Program sa skladá z jedného veľkého hlavného

modulu. Tento modul sa nazýva **Simulator Frame**, ktorý sa delí na menšie podmoduly. Medzi podmoduly patrí menu bar Simulatoru, grafický panel pre vykreslenie simulácie, testovací panel algoritmu, ktorý simuluje kroky algoritmu a textová oblasť algoritmu, do ktorej algoritmus pridá textový formát súčasne vytvorených nových plánov. Aby tieto moduly mohli spolu komunikovať vytvoril som jeden menší modul, cez ktorý moduly komunikujú, tento modul som nazval **Databáza**. Implementácia modulu sa nachádza v zdrojovom kóde **SimulatorDatabase.java.** Zdrojové kódy programu som rozdelil do štyroch väčších skupín.

Prvá skupina sa nazýva **Simulator data structures** a nachádza sa v balíku (*package)* **SimulatorDataStructures**, ktorý môžeme nájsť v adresári **Simulator/src.**

U každého zdrojového kódu som okomentoval funkcie, že akú operáciu vykonávajú. Z komentára sa dá vygenerovať **Javadoc**.

Druhá skupina sa nazýva **Simulator** a nachádza sa v adresári **Simulator/src**.

V tomto adresári sa nachádza grafika programu a hlavné moduly programu a implementácie funkcionality programu.

Tretia skupina sa nazýva **AlgorithmBase** a nachádza sa v adresári **Simulator/src.**

V tomto adresári je abstraktný predok algoritmov od ktorého musia dediť všetky algoritmy a musia implementovať funkciu **run**. V tomto adresári ešte sa nachádzajú pomocné dátové štruktúry implementovaných algoritmov.

Posledná skupina sa nazýva **Algorithm** a nachádza sa v adresári **Simulator/src.**

V tomto adresári sa nachádzajú algoritmy, ktoré použijeme na vytvorenie rozvrhu kamiónov v simulácií.

5.4.1 Dátové štruktúry programu Simulator

Už vieme, že zdrojové kódy sú rozdelené do štyroch skupín. Z prvej skupiny by som chcel zdôrazniť najdôležitejšie triedy (**class**), ktoré tvoria základ celej aplikácie. Medzi tieto triedy patrí trieda **SynchronizedCity**, trieda **SynchronizedRoad**, trieda **SynchronizedTruck**, trieda **SynchronizedPackage,** trieda **ShortestPath**, trieda **Plan,** trieda **TruckState** a trieda **SimulatorDatabase**. Implementáciu týchto tried môžeme nájsť v súboroch **SynchronizedCity.java, SynchronizedRoad.java, SynchronizedTruck.java, SynchronizedPackage.java, ShortestPath.java, Plan.java, TruckState.java** a **SimulatorDatabase.java**.

Trieda **SynchronizedCity** modeluje všetky vlastnosti mesta, **SynchronizedRoad** modeluje orientovanú cestu, **SynchronizedTruck** modeluje všetky dôležité vlastnosti kamiónu, ktoré sú používané pri simulácií. Trieda **SynchronizedPackage** modeluje vlastnosti objednávky, trieda **ShortestPath** obsahuje poľe ciest, ktoré reprezentujú najkratšiu cestu medzi dvoma rôznymi mestami. Trieda **Plan** reprezentuje rozvrh medzi kamiónom a objednávkou, ktorú obslúži. Trieda **TruckState** popíše stav kamiónu, ktorý súčasne obslúži nejakú objednávku, obsahuje identifikačné číslo cesty na ktorom sa nachádza a časť cesty, ktorú už absolvoval, táto hodnota je z intervalu [0, 1]. Trieda **SimulatorDatabase** obsahuje všetky mestá, cesty, kamióny a objednávky, zaistí rýchly prístup k objektom v databáze. K cestám, mestám a najkratším cestám v databáze sa dá dostať v $O(1)$ čase a ku kamiónom a objednávkam v $O(\log n)$, kde n je počet kamiónov alebo objednávok. Všetky tieto triedy sú chránené proti vícenásobnému prístupu vlákien (**thread safe**).

V druhej skupine najdôležitejšie triedy sú **SimulatorFrame**, **GraphicalPanel**, **DebugPanel, StartSimulationMenuItem** a **SimulatorStepCalculator**. **SimulatorFrame** obsahuje implementáciu hlavného okienka aplikácie a dedí svoje vlastnosti od triedy **JFrame**.Trieda **GraphicalPanel** má na starosti vykresliť sieti ciest a miest a vykresliť pohybujúce kamióny, ktoré obslúžia objednávky. Rozvrhovací algoritmus komunikuje s grafickým panelom a s testovacím panelom pomocou udalosti (event). Trieda **DebugPanel** má na starosti vykresliťcykly rozvrhovacieho algoritmu. Triedy DebugPanel a GraphicalPanel dedia svoje základné vlastnosti od triedy **JPanel**. Komunikáciu pomocou udalostí potrebujeme kvôli tomu , aby virtuálny stroj Javy neoptimalizoval prekreslenie panelov. Pomocou triedy **StartSimulatorMenuItem** môžeme vybrať rozvrhovací algoritmus. Trieda otestuje či súbor triedy, ktorý sme si vybrali je potomkom triedy Algorithm, ak nie, nahlási chybu, ak áno zavolá metódu **run**.

Trieda **SimulatorStepCalculator** vypočíta nové stavy kamiónov, ktoré súčasne obsluhujú objednávky. Po každom cykle výpočtu vytvorí udalosť, ktorú posiela grafickému panelu.

V tretej skupine je predok algoritmov, ktorý implementuje funkcionality, ktoré sú spoločné pre všetky algoritmy. Medzi tieto funkcionality patrí filtrovanie kamiónov, ktoré nemôžu obslúžiť žiadnu objednávku a filtrovanie objednávok, ktoré nemôžu byť obslúžené.

V poslednej skupine sú implementácie algoritmov, ktoré sú popísané v tretej kapitole.
Na konci programátorskej dokumentácie by som chcel ešte pripomenúť dve pokročilé techniky jazyka Java, ktoré som použil v mojom programe Simulator.

5.4.2 Java Reflection API, Dynamic class loading

Reflection API (http://docs.oracle.com/javase/tutorial/reflect/index.html)
a **Dynamic class loading** patria do skupiny vyšších programovacích techník programovacieho jazyka **Java**. Pomocou dynamic class loading môžeme dynamicky načítať triedy pri behu programu. Reflection API nám dovolí získať informácie o načítanej triede. Pomocou Reflection API môžeme získať všetky metódy danej triedy aj tie, u ktorých nemáme prístup pri programovaní. Je to veľmi mocný nástroj, ale keď chceme efektívne použiť a nechceme, aby program vykonával nepredvídateľné operácie musíme dávať veľký pozor.
Tieto techniky používam pri pridaní rozvrhovacieho algoritmu k programu Simulator.

Záver

Zhodnotenie

V práci sa nám podarilo popísať všetky algoritmy, ktoré boli následne implementované. Vyvinuli sme systém pracujúci s implementovanými algoritmami. Náš systém je univerzálny, nakoľko pridanie ďalších algoritmov nevyžaduje zmenu základov systému. Pri pridaní ďalšieho algoritmu podmienkou je len, že trieda nového algoritmu musí byť potomkom triedy **Algorithm.** Rozvrhovacie cykly algoritmov môžeme vizualizovať v systéme pomocou **Debug Mode**.
Cieľom práce bolo porovnanie jednotlivých typov rozvrhovacích algoritmov. Cieľ práce bol splnený.

Možnosti rozšírenia

Prácu je možné rozšíriť v oblasti grafiky, vo vylepšení plug-in systému, pridaním testovacieho modulu, zlúčením programu Editor a Simulator do spoločného programu.

Zoznam použitej literatúry

[1] Martin Christopher - Logistics & Supply Chain Management

[2] Michael R. Garey / David S. Johnson - Computers and Intractability ; A Guide to the Theory of NP-Completeness,1990

[3] Thomas Cormen, Charlese Leiserson, Ronald L. Rivest, Clifford Stein - Introduction to Algorithms

[4] Massimo Paolucci, http://www.discovery.dist.unige.it/didattica/LS/VRP.pdf ,

[5] R. Barták, prednášky ,http://ktiml.mff.cuni.cz/~bartak/podminky/index.html

[6] J. Matoušek : Lineární programování a lineární algebra pro informatiky,2006.

[7] Dan Gusfield and Robert W. Irving, The Stable Marriage Problem, Structure and Algorithms,

[8] Jiří Anděl : Statistické metody

Zoznam použitých skratiek

VRP - Vehicle routing problem – dopravný problém

CSP - Constraint satisfaction problem- modelovanie problémov s podmienkami

NRP - Node routing problem - uzlovo dopravný problém

ARP - Arc routing problem - hranovo dopravný problém

TSP - Problém obchodných cestujúcich

TSPB - Problém obchodných cestujúcich so spiatočnou cestou

TSPTW - Problém obchodných cestujúcich s časovým okienkom

MTSP - Multi - TSP

CVRP - Kapacitný dopravný problém

VRPB - Dopravný problém so spiatočnou cestou

DCVRP - Dopravný problém s podmienkami na vzdialenosť

VRPTW - Dopravný problém s časovým okienkom

VRPPD - Dopravný problém VRP s naložením a dodaním

LP - Lineárne programovanie

LC - Linehaul customer - zákazník potrebuje požadované množstvo tovaru dodané

BC - Backhaul customer – zákazník potrebuje požadované množstvo tovaru naložiť